

Brian W.
Kernighan
Dennis M.
Ritchie



A LINGUAGEM DE PROGRAMAÇÃO

EDISA
Editora Campus

VISIT...

LANZAROTE
Caliente.COM

TÍTULOS DE INTERESSE CORRELATO

C

C: O Livro de Respostas — *C.L. Tondo e S.E. Gimpel*
DO BASIC AO C — *R.J. Traister*
MANUAL DE LINGUAGEM C — *L. Hancock e M. Krieger*
LINGUAGEM C: Guia de Referência — *F. Cabral*
C AVANÇADO: Técnicas e Truques — *G.E. Sobelman e D.E. Krekelberg*
DEPURANDO EM C — *R. Ward*
A LINGUAGEM C E O PC BIOS — *F. Cabral*
C: Quick Reference — *A.C. Plantz*
C: A Linguagem de Programação Padrão ANSI — *B.W. Kernighan e D.M. Ritchie*

Solicite nosso catálogo completo.

Procure nossas publicações nas boas livrarias ou comunique-se diretamente com:

EDITORA CAMPUS LTDA.

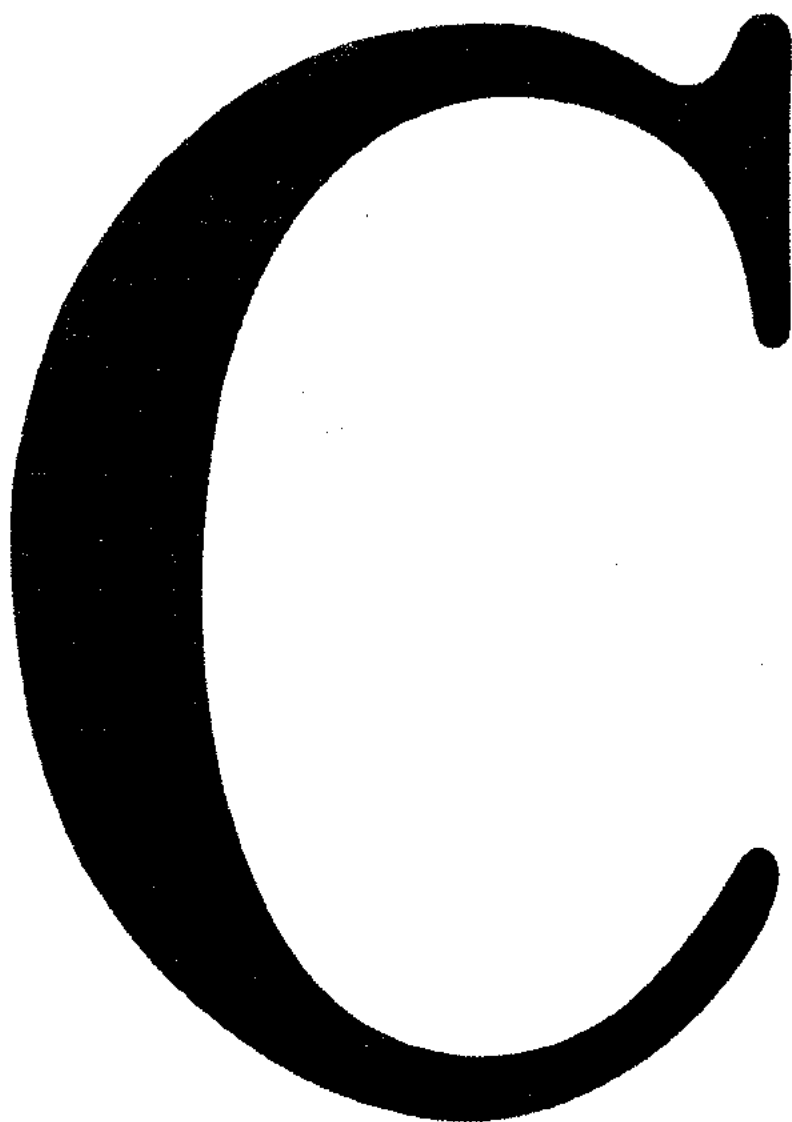
Qualidade internacional a serviço do autor e do leitor nacional.

Rua Barão de Itapagipe, 55 Rio Comprido

Tel. PABX (021) 293-6443 Telex (021) 32606 EDCP BR

20261 Rio de Janeiro RJ Brasil

Endereço Telegráfico: CAMPUSRIO



A LINGUAGEM DE PROGRAMAÇÃO

Brian W.
Kernighan
Dennis M.
Ritchie

Bell Laboratories
Murray Hill, New Jersey

6ª Reimpressão

A LINGUAGEM DE PROGRAMAÇÃO



EDISA
Eletrônica Digital S/A
Porto Alegre

Editora Campus Ltda.

Do original:
The C Programming Language.
Original edition in english language published by Prentice-Hall, Inc.
Copyright © 1978. All Rights Reserved.
© 1986, Editora Campus Ltda.
6ª Reimpressão, 1989.

Todos os direitos reservados e protegidos pela Lei 5988 de 14/12/73.
Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Todo o esforço foi feito para fornecer a mais completa e adequada informação.
Contudo a editora e o(s) autor(es) não assumem responsabilidade pelos resultados e uso da informação fornecida. Recomendamos aos leitores testar a informação antes de sua efetiva utilização.

Capa
Otavio Studart

Tradução
Pedro Sérgio Nicoletti
Infocon — Assessoria e Consultoria
em Informática e Teleinformática Ltda.
com apoio da Edisa

Projeto Gráfico, Composição e Revisão
Editora Campus Ltda.
Qualidade internacional a serviço do autor e do leitor nacional.
Rua Barão de Itapagipe 55 Rio Comprido
Telefone: (021) 293 6443 Telex: (021) 32606 EDCP BR
20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

ISBN 85-7001-410-4
[Edição original: ISBN 0-13-110163-3, Prentice-Hall, Inc., New Jersey, USA.]

Ficha Catalográfica
CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ.

K47c Kernighan, Brian W.
C: a linguagem de programação / Brian W. Kernighan, Dennis M. Ritchie; tradução sob a responsabilidade da EDISA - Eletrônica Digital S.A. — Rio de Janeiro: Campus; Porto Alegre: Edisa, 1986.

Tradução de: The C Programming Language.
Apêndice.
ISBN 85-7001-410-4

1. C (Linguagem de programação para computadores). I. Ritchie, Dennis M. II. Título.

86-1123

CDD — 001.6424
CDU — 800.92C

PREFÁCIO

A 1ª EDIÇÃO BRASILEIRA

Empresas inovadoras que chamam a si a missão de desbravar novos mercados, com tecnologia avançada, enfrentam sempre dois grandes desafios.

O primeiro é intrínseco ao pioneirismo. Elas têm que acreditar em seus produtos e correr os riscos e as incertezas que acompanham todas as inovações.

O segundo desafio está em implantar, nas suas áreas de atuação, uma nova cultura tecnológica que venha a se constituir no ambiente propício para a evolução mercadológica de seus produtos.

Em consequência, cria-se um processo sinérgico onde há um crescimento cultural tanto das empresas como das comunidades com as quais elas interagem.

Tendo a EDISA assumido a vanguarda no mercado brasileiro de supermicros baseados em sistemas operacionais de padrão internacional com o lançamento da linha ED-600 em 1984, é natural que por isso também assumisse responsabilidades junto à comunidade nacional de informática.

Foi dentro deste espírito que surgiu a iniciativa de oferecer aos técnicos brasileiros uma edição em português do clássico "THE C PROGRAMMING LANGUAGE", de BRIAN W. KERNIGHAN e DENNIS M. RITCHIE.

Com esta edição, mais do que simplesmente preencher uma lacuna existente na bibliografia em português de informática, está sendo dado um forte impulso para que se evolua no esforço comum de criar uma avançada tecnologia nacional.

Associadas à EDISA neste trabalho estão a INFOCON, uma das empresas pioneiras no Brasil na pesquisa e desenvolvimento de softwares em linguagem C, que realizou a tradução, e a Editora CAMPUS, que mais uma vez diz presente quando chamada a colaborar com a área de informática.

A todos os nossos agradecimentos.

EDISA – Eletrônica Digital S/A

PREFÁCIO

C é uma linguagem de programação de finalidade geral que permite economia de expressão, modernos fluxos de controle e estruturas de dados e um rico conjunto de operadores. C não é uma linguagem de "muito alto nível", nem "grande" e nem específica para uma área de aplicação particular. Mas, sua falta de restrições e sua generalidade tornam-na mais conveniente e eficaz para muitas tarefas do que linguagens supostamente mais poderosas.

C foi originalmente projetada para ser implementada no sistema operacional UNIX* no PDP-11 da DEC (Digital Equipment Corporation), por Dennis Ritchie. O sistema operacional, o compilador C e essencialmente todos os programas de aplicação do UNIX (incluindo todo o software usado para preparar a presente tradução) são escritos em C. Compiladores de produção também existem para várias outras máquinas, incluindo o IBM System/370, Honeywell 6000, e o Interdata 8/32. C, entretanto, não é ligada a nenhum hardware ou sistema particular, e é fácil escrever programas que rodarão sem mudanças em qualquer máquina que aceite a linguagem C.

Este livro propõe-se a auxiliar o leitor a aprender como programar em C. Ele contém uma introdução através de exemplos para que usuários novatos possam ser iniciados o mais cedo possível, capítulos separados para cada uma das características principais, e um manual de referência. A maioria do tratamento é baseada na leitura, escrita e revisão de exemplos, ao invés de meras definições de regras. Na maior parte, os exemplos são programas reais completos, ao invés de fragmentos isolados. Todos os exemplos foram testados diretamente a partir do texto, que existe em forma legível por máquina**. Apesar de mostrar como fazer uso efetivo da linguagem, também tentamos, onde possível, ilustrar algoritmos úteis e princípios de bom estilo e bom projeto.

O livro não é um manual introdutório de programação; ele presume alguma familiaridade com conceitos básicos de programação tais como variáveis, comandos de atribuição, laços, e funções. No entanto, um programador novato deve ser capaz de lê-lo e aprender a linguagem, embora o contato com um colega com mais conhecimento poderá ser útil.

Em nossa experiência, C tem provado ser uma linguagem agradável, expressiva, e versátil para uma grande gama de programas. Sua aprendizagem é fácil e ela responde bem à medida que a experiência com ela cresce. Esperamos que este livro o ajude a usá-la bem.

* UNIX é uma marca registrada dos Bell Laboratories. O sistema operacional UNIX é disponível sob licença da Western Electric, Greensboro, N.C.

** Isto não se aplica à presente tradução. (N. do T.)

As críticas construtivas e sugestões de amigos e colegas acrescentaram muito a este livro e ao nosso prazer em escrevê-lo. Em particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin, e Larry Rosler leram múltiplas versões com cuidado. Nós também devemos muito a Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson e Peter Weinberger por comentários úteis em vários estágios do trabalho e a Mike Lesk e Joe Ossanna pela assistência incalculável na impressão da versão em inglês.

Brian W. Kernighan
Dennis M. Ritchie

INTRODUÇÃO

C é uma linguagem de programação de finalidade geral. Ela é estreitamente associada ao sistema operacional UNIX já que foi desenvolvida neste sistema, e já que UNIX está escrito em C. A linguagem, entretanto, não é atada a um sistema operacional ou a uma máquina particulares, e, embora tenha sido chamada "linguagem de programação de software básico" devido à sua utilidade no desenvolvimento de sistemas operacionais, ela tem sido usada, igualmente, para escrever grandes programas numéricos, de processamento de texto, e bancos de dados.

C é uma linguagem de relativo "baixo nível". Esta caracterização não a desmerece; isto simplesmente significa que C manipula o mesmo tipo de objetos que a maioria dos computadores, tais como caracteres, números, e endereços. Estes podem ser combinados e manipulados com os operadores aritméticos e lógicos usuais implementados pelas máquinas atuais.

C não provê operações para manipular diretamente objetos compostos tais como cadeias de caracteres, conjuntos, listas, ou arranjos considerados como um todo. Não há equivalente, por exemplo, para operações do PL/1 que manipulam um arranjo ou uma cadeia por completo. A linguagem não define nenhuma facilidade para alocação de memória outra que a definição estática e a disciplina de pilha fornecidas pelas variáveis locais de funções; não há monte ou coleta de lixo como encontrados no Algol 68. Finalmente, C não provê facilidades de entrada e saída: não há comandos READ ou WRITE, nem métodos de acesso a arquivos. Todos esses mecanismos devem ser fornecidos por funções explicitamente chamadas.

De forma semelhante, C oferece somente construções simples de fluxo de controle: testes, laços, agrupamentos e subprogramas, mas não multiprogramação, operações paralelas, sincronização, ou co-rotinas.

Embora a falta de algumas destas facilidades possa parecer uma grave deficiência ("Você quer dizer que eu tenho que chamar uma função para comparar duas cadeias de caracteres?"), a manutenção da linguagem em dimensões modestas tem trazido benefícios reais. Desde que C é relativamente pequena, ela pode ser descrita em pouco espaço, e aprendida rapidamente. Um compilador C pode ser simples e compacto. Compiladores são também facilmente escritos; usando a tecnologia atual, pode-se esperar um prazo de dois meses para escrever um compilador para uma máquina nova, e verificar que 80% do código do novo compilador são comuns aos já existentes. Isto provê um alto grau de mobilidade de linguagem. Como os tipos de dados e estruturas de controle providos por C são suportados diretamente pela maioria dos computadores existentes, o ambiente de supor-

te para implementar programas autocontidos é pequeno. No PDP-11, por exemplo, o ambiente requer somente as rotinas que fazem multiplicação e divisão em 32 bits e para desenvolver a sequência de entrada e saída de sub-rotinas. Claro que cada implementação provê uma biblioteca compreensível e compatível de funções que desenvolvem operações de entrada e saída, manipulação de cadeias e alocação de memória, mas, desde que estas funções são chamadas explicitamente, podem ser evitadas se necessário; podendo também ser escritas de forma transportável na própria linguagem C.

Novamente, como a linguagem reflete a capacidade dos computadores atuais, programas em C tendem a ser eficientes o bastante para que não haja inclinação para escrever programas em linguagem de montagem. O exemplo mais óbvio disso é o próprio sistema operacional UNIX, que está escrito quase que totalmente em C. De 13000 linhas de código, apenas cerca de 800 linhas estão escritas em linguagem de montagem. Além disso, essencialmente todo software de aplicação do UNIX está escrito em C. A vasta maioria de usuários do UNIX (incluindo um dos autores deste livro) nem sequer conhece a linguagem de montagem do PDP-11.

Embora C explore as capacidades de muitos computadores, é independente da arquitetura de qualquer máquina particular; assim, com um pouco de cuidado, é fácil escrever programas "transportáveis", isto é, programas que podem ser executados em uma variedade de máquinas sem nenhuma alteração. É agora comum em nosso ambiente que o software desenvolvido no UNIX seja transportado para os sistemas Honeywell, IBM e Interdata. De fato, os compiladores C e as bibliotecas de execução dessas 4 máquinas são muito mais compatíveis que as supostas versões padrão ANSI do Fortran. O sistema operacional UNIX roda hoje no PDP-11 e no Interdata 8/32. Fora os programas que são necessariamente dependentes de máquinas tais como compilador, montador, e depurador, o software escrito em C é idêntico em ambas as máquinas. No sistema operacional propriamente dito, 7000 linhas de código, fora o suporte de linguagem de montagem e os manipuladores de dispositivos de entrada e saída, são aproximadamente 95% idênticas.

Para programadores familiarizados com outras linguagens, é útil mencionar alguns aspectos históricos, técnicos, e filosóficos da linguagem C para efeito de contraste e comparação.

Muitas das idéias importantes de C vêm da consideravelmente mais velha, mas ainda vital, linguagem BCPL, desenvolvida por Martin Richards. A influência de BCPL em C procede indiretamente pela linguagem B, que foi escrita por Ken Thompson em 1970 para o primeiro sistema UNIX no PDP-7.

Embora C compartilhe muitas facilidades características com BCPL, C não é, em nenhum sentido, um dialeto dela. BCPL e B são linguagens "sem tipo": o único tipo de dado é a palavra de máquina, e o acesso a outros tipos de objetos é feito por operadores especiais ou chamadas de função. Em C, os objetos de dados fundamentais são caracteres, inteiros de vários tamanhos, e números com ponto flutuante. Além disso, há uma hierarquia de tipos de dados derivados criados com apontadores, arranjos, estruturas, uniões, e funções.

C provê as construções fundamentais de fluxo de controle necessárias para programas bem estruturados: agrupamentos de comandos; tomada de decisão (if); laços com o teste de encerramento no início (while, for) ou no fim (do); e seleção de um dentre um conjunto de possíveis casos (switch). (Tudo isso era encontrado em BCPL, embora com algumas diferenças sintáticas; esta linguagem antecipou a moda de "programação estruturada" em muitos anos.)

C provê apontadores e a habilidade de fazer aritmética com endereços. Os argumentos para funções são passados copiando-se o valor do argumento, e é impossível para a função chamada mudar o valor atual do argumento na função que a chamou. Quando é necessário realizar uma "chamada por referência", um apontador pode ser passado explicitamente, e a função pode mudar o objeto para o qual aponta o apontador. Nomes de arranjos são passados como a localização da origem do arranjo, de forma que argumentos do tipo arranjo são efetivamente chamados por referência.

Qualquer função pode ser chamada recursivamente, e suas variáveis locais são normalmente "automáticas", ou seja, criadas novamente a cada ativação. Definições de funções não podem ser aninhadas, mas as variáveis podem ser declaradas seguindo uma estrutura de bloco. As funções de um programa C podem ser compiladas separadamente. Variáveis podem ser internas a uma função, externas mas conhecidas apenas em um arquivo fonte, ou completamente globais. Variáveis internas podem ser automáticas ou estáticas. Variáveis automáticas podem ser colocadas em registradores para aumentar a eficiência, mas a declaração de registrador é apenas uma dica para o compilador, e não se refere a registradores específicos das máquinas.

C não é uma linguagem com verificação rigorosa de tipos no sentido de Pascal ou Algol 68. Ela é relativamente permissiva na conversão de dados, embora não converta automaticamente tipos de dados com o descaso do PL/1. Compiladores existentes não provêem verificação de subíndices de arranjos, tipos de argumentos, etc., em tempo de execução.

Para situações onde a verificação de tipos é desejável, uma versão separada do compilador chamada *lint* é usada. *Lint* não gera código, mas aplica uma rigorosa verificação a tempo de compilação e de carga. Ele detecta incompatibilidade de tipos, uso inconsistente de argumentos, variáveis não usadas ou aparentemente não inicializadas, dificuldades potenciais de transportabilidade, e coisas similares. Programas que passam incólumes pelo *lint*, com raras exceções, são tão isentos de erros de tipos quanto, por exemplo, programas em Algol 68. Mencionaremos outras capacidades do *lint* oportunamente.

Finalmente, C, como qualquer outra linguagem, tem suas falhas. Alguns operadores têm a precedência errada; algumas partes da sintaxe poderiam ser melhores; há várias versões da linguagem em existência, diferindo levemente. No entanto, C tem provado ser uma linguagem extremamente expressiva e efetiva para uma grande variedade de aplicações.

O resto do livro é organizado como segue. O Capítulo 1 é uma introdução à parte central de C através de exemplos. A finalidade é fazer com que o leitor se inicie tão rápido quanto possível, pois acreditamos firmemente que a única maneira de aprender uma nova linguagem é escrevendo programas nela. A introdução presume um conhecimento prático dos elementos básicos de programação; não há explicações sobre computadores, compiladores, ou o significado de uma expressão do tipo $n = n + 1$. Embora tenhamos tentado, onde possível, mostrar técnicas úteis de programação, o livro não objetiva ser um trabalho de referência em estruturas de dados e algoritmos; quando forçados a uma escolha nós nos concentramos na linguagem.

Os Capítulos 2 a 6 discutem vários aspectos de C mais detalhadamente e um pouco mais formalmente que o Capítulo 1, embora a ênfase seja ainda em exemplos de programas úteis, completos, ao invés de fragmentos isolados. O Capítulo 2 lida com os tipos de dados básicos, operadores e expressões. O Capítulo 3 trata de fluxo de controle: *if-else*, *while*, *for*, etc. O Capítulo 4 cobre funções e estruturas de programa — variáveis externas,

regras de escopo, e assim por diante. O Capítulo 5 discute apontadores e aritmética com endereços. O Capítulo 6 contém detalhes de estruturas e uniões.

O Capítulo 7 descreve a biblioteca padrão de entrada e saída em C, que provê uma interface comum para o sistema operacional. Esta biblioteca de entrada e saída é suportada em todas as máquinas que rodam C, de forma que programas que usam-na para entrada, saída e outras funções do sistema podem ser transportados de um sistema para outro essencialmente sem mudança.

O Capítulo 8 descreve a interface entre programas C e o sistema operacional UNIX, concentrando-se em entrada e saída, sistema de arquivos, e transportabilidade. Embora uma parte deste capítulo seja específica para o UNIX, programadores que não estejam usando UNIX devem ainda encontrar material útil nele, inclusive alguma luz sobre como uma versão da biblioteca padrão é implementada e sugestões para obter código transportável.

O Apêndice A contém o manual de referência da linguagem C. Ele é o comunicado "oficial" de sintaxe e semântica de C, e (com exceção do compilador que você esteja usando), é o árbitro final sobre quaisquer ambiguidade e omissões dos capítulos anteriores.

Desde que C é uma linguagem que envolve, e existe em, uma variedade de sistemas, alguma parte do material deste livro pode não corresponder ao estado corrente do desenvolvimento de um sistema particular. Tentamos evitar estes problemas e apontar dificuldades potenciais. Em caso de dúvida, entretanto, escolhemos descrever de forma generalizada a situação no UNIX do PDP-11, desde que é o ambiente da maioria dos que programam em C. O Apêndice A também descreve diferenças de implementação nos principais sistemas C.

SUMÁRIO

Capítulo 1	Uma Introdução Através de Exemplos	19
1.1	Início	19
1.2	Variáveis e Aritmética	21
1.3	O Comando For	24
1.4	Constantes Simbólicas	25
1.5	Uma Coleção de Programas Úteis	26
1.6	Arranjos	32
1.7	Funções	34
1.8	Argumentos – Chamada por Valor	35
1.9	Arranjos de Caracteres	36
1.10	Escopo; Variáveis Externas	39
1.11	Sumário	41
Capítulo 2	Tipos, Operadores e Expressões	42
2.1	Nomes de Variáveis	42
2.2	Tipos de Dados e Tamanhos	42
2.3	Constantes	43
2.4	Declarações	45
2.5	Operadores Aritméticos	46
2.6	Operadores Relacionais e Lógicos	46
2.7	Conversões de Tipos	47
2.8	Operadores de Incremento e Decremento	50
2.9	Operadores Lógicos Bit-a-Bit	52
2.10	Operadores e Expressões de Atribuição	53
2.11	Expressões Condicionais	55
2.12	Precedência e Ordem de Avaliação	56
Capítulo 3	Fluxo de Controle	58
3.1	Comandos e Blocos	58
3.2	If-Else	58

3.3	Else-If	59
3.4	Switch	60
3.5	Laços – While e For	62
3.6	Laços – Do-while	65
3.7	Break	66
3.8	Continue	67
3.9	Goto e Rótulos	67
Capítulo 4	Funções e Estrutura de um Programa	69
4.1	Conceitos Básicos	69
4.2	Funções que retornam Valores não Inteiros	72
4.3	Mais sobre Argumentos de Funções	74
4.4	Variáveis Externas	74
4.5	Regras de Escopo	78
4.6	Variáveis Estáticas	81
4.7	Variáveis em Registradores	82
4.8	Estrutura de Bloco	83
4.9	Inicialização	84
4.10	Recursividade	85
4.11	O Preprocessador C	87
Capítulo 5	Apontadores e Arranjos	89
5.1	Apontadores e Endereços	89
5.2	Apontadores e Argumentos de Funções	91
5.3	Apontadores e Arranjos	93
5.4	Aritmética com Endereços	95
5.5	Apontadores de Caractere e Funções	98
5.6	Apontadores Não São Inteiros	100
5.7	Arranjos Multidimensionais	101
5.8	Arranjos de Apontadores; Apontadores para Apontadores	103
5.9	Inicialização de Arranjos de Apontadores	106
5.10	Apontadores <i>Versus</i> Arranjos Multidimensionais	106
5.11	Argumentos da Linha de Comando	107
5.12	Apontadores para Funções	111
Capítulo 6	Estruturas	114
6.1	Elementos Básicos	114
6.2	Estruturas e Funções	116
6.3	Arranjos de Estruturas	118
6.4	Apontadores para Estruturas	121
6.5	Estruturas Auto-Referenciadas	123
6.6	Pesquisa em Tabela	127
6.7	Campos	129

6.8	Unões	130
6.9	Typedef	132
Capítulo 7	Entrada e Saída	134
7.1	Acesso à Biblioteca Padrão	134
7.2	Entrada e Saída Padrão – Getchar e Putchar	135
7.3	Saída Formatada - Printf	136
7.4	Entrada Formatada - Scanf	137
7.5	Conversão de Formato na Memória	140
7.6	Acesso a Arquivos	140
7.7	Tratamento de Erro – Stderr e Exit	143
7.8	Entrada e Saída de Linhas	144
7.9	Algumas Funções Diversas	145
Capítulo 8	A Interface com o Sistema UNIX⁺	147
8.1	Descritores de Arquivos	147
8.2	Entrada e Saída de Baixo Nível - Read e Write	148
8.3	Open, Creat, Close, Unlink	149
8.4	Acesso Randômico - Seek e Lseek	151
8.5	Exemplo – Uma Implementação de Fopen e Getc	152
8.6	Exemplo – Listagem de Diretórios	155
8.7	Exemplo – Um Alocador de Memória	159
Apêndice A	Manual de Referência C	163
1.	Introdução	163
2.	Convenções Léxicas	163
2.1	Comentários	163
2.2	Identificadores (Nomes)	163
2.3	Palavras-Chaves	164
2.4	Constantes	164
2.4.1	Constantes Inteiras	164
2.4.2	Constantes Longas Explícitas	164
2.4.3	Constantes do Tipo Caractere	164
2.4.4	Constantes do Tipo Ponto Flutuante	165
2.5	Cadeias	165
2.6	Características de Hardware	165
3.	Notação Sintática	166
4.	Em que Consiste um Nome?	166
5.	Objetos e lvalues	167
6.	Conversões	167
6.1	Caracteres e Inteiros	167
6.2	Float e Double	167
6.3	Ponto Flutuante e Integral	167

6.4	Apontadores e Inteiros	168
6.5	Unsigned	168
6.6	Conversões Aritméticas	168
7.	Expressões	168
7.1	Expressões Primárias	169
7.2	Operadores Unários	170
7.3	Operadores Multiplicadores	172
7.4	Operadores Adicionadores	172
7.5	Operadores de Deslocamento	173
7.6	Operadores Relacionais	173
7.7	Operadores de Igualdade	173
7.8	Operador e Bit-a-Bit	174
7.9	Operador ou-exclusivo Bit-a-Bit	174
7.10	Operador ou-inclusivo Bit-a-Bit	174
7.11	Operador e Lógico	174
7.12	Operador ou Lógico	174
7.13	Operador Condicional	174
7.14	Operadores de Atribuição	175
7.15	Operador Vírgula	175
8.	Declarações	176
8.1	Especificadores de Classe de Armazenamento	176
8.2	Especificadores de Tipo	177
8.3	Declaradores	177
8.4	Significado de Declaradores	178
8.5	Declarações de Estrutura e União	179
8.6	Inicialização	181
8.7	Nomes de Tipos	183
8.8	Typedef	183
9.	Comandos	184
9.1	Comando de Expressão	184
9.2	Comando Composto ou Bloco	184
9.3	Comando Condicional	184
9.4	Comando While	185
9.5	Comando Do	185
9.6	Comando For	185
9.7	Comando Switch	185
9.8	Comando Break	186
9.9	Comando Continue	186
9.10	Comando Return	186
9.11	Comando Goto	186
9.12	Comando Rotulado	187
9.13	Comando Nulo	187
10.	Definições Externas	187
10.1	Definições de Funções Externas	187

10.2	Definições de Dados Externos	188
11.	Regras de Escopo	188
11.1	Escopo Léxico	189
11.2	Escopo de Objetos Externos	189
12.	Linhas de Controle do Compilador	190
12.1	Reposição de Símbolos	190
12.2	Inclusão de Arquivo	190
12.3	Compilação Condicional	191
12.4	Controle de linha	191
13.	Declarações Implícitas	191
14.	Tipos Reconsiderados	191
14.1	Estruturas e Uniões	192
14.2	Funções	192
14.3	Arranjos, Apontadores e Indexação	192
14.4	Conversões Explícitas de Apontadores	193
15.	Expressões Constantes	194
16.	Considerações de Transportabilidade	194
17.	Anacronismos	195
18.	Sumário da Sintaxe	196
18.1	Expressões	196
18.2	Declarações	197
18.3	Comandos	199
18.4	Definições Externas	200
18.5	Pré-processador	200

Índice Analítico	201
-------------------------	------------

Capítulo 1

UMA INTRODUÇÃO ATRAVÉS DE EXEMPLOS

Vamos começar com uma rápida introdução a C. Nosso objetivo é mostrar os elementos essenciais da linguagem em programas reais, mas sem esmiuçar detalhes, regras formais, e exceções. Neste ponto, não estamos tentando ser completos ou mesmo precisos (salvo os exemplos, que são corretos). Queremos levá-lo, tão rapidamente quanto possível, ao ponto onde você possa escrever programas úteis; devemos, portanto, nos concentrar nos elementos básicos: variáveis e constantes, aritmética, fluxo de controle, funções e os rudimentos de entrada e saída. Intencionalmente deixamos de fora neste capítulo facilidades de C que são de vital importância para escrever programas maiores. Isto inclui apontadores, estruturas, a maior parte do rico conjunto de operadores C, vários comandos de fluxo de controle, e uma centena de detalhes.

Este enfoque tem suas desvantagens, é claro. O mais notável é que o tratamento completo de algum aspecto da linguagem não é encontrado num único local, e o capítulo, por ser breve, pode também confundir. E já que os exemplos não podem usar o poder total de C, eles não são tão concisos e elegantes quanto poderiam ser. Tentamos minimizar esses efeitos, mas esteja prevenido.

Uma outra desvantagem é que os próximos capítulos necessariamente repetirão alguma parte deste.

Em todo caso, programadores experientes devem ser capazes de extrapolar o material deste capítulo para suas próprias necessidades de programação. Iniciantes deveriam suplementá-lo, tentando pequenos programas, similares aos deste capítulo. Ambos os grupos podem usá-lo como infra-estrutura na qual se acoplam as descrições mais detalhadas que iniciamos no Capítulo 2.

1.1 Início

O único caminho para aprender uma nova linguagem de programação é escrever programas nesta linguagem. O primeiro programa a escrever é o mesmo em todas as linguagens:

Imprima as palavras

primeiro programa

Este é o obstáculo básico: para saltá-lo você deve ser capaz de criar o texto do programa em algum lugar, compilá-lo com sucesso, carregá-lo, executá-lo e ver para onde sua resposta foi. Uma vez aprendidos estes detalhes mecânicos, tudo o mais é relativamente fácil.

Em C, o programa para imprimir “primeiro programa” é:

```
main ( )  
{  
    printf (“primeiro programa\n”);  
}
```

O modo de executar este programa depende do sistema que você está usando. Como um exemplo específico, no sistema operacional UNIX você deve criar o programa fonte num arquivo cujo nome termine em “.c”, tal como *primeiro.c* e então compilá-lo com o comando

cc primeiro.c

Se você não errar nada, tal como omitir um caractere ou escrever algo errado, a compilação transcorrerá silenciosamente, e produzirá um arquivo executável chamado *a.out*. Executando-o com o comando

a.out

produzirá como saída:

primeiro programa

Em outros sistemas, as regras serão diferentes; verifique com um especialista.

Exercício 1-1. Execute este programa no seu sistema. Experimente deixar de fora partes do programa para ver que mensagem de erro você obtém.

Agora, algumas explicações sobre o programa. Um programa C, independentemente de seu tamanho, consiste de uma ou mais “funções” que especificam as operações computacionais que devem ser feitas. Funções C são similares a funções e sub-rotinas de um programa Fortran, ou os procedimentos de PL/1, Pascal etc. No nosso exemplo, main é uma função. Normalmente, você poder escolher o nome que gostar, mas main é um nome especial — seu programa começa a ser executado no início de main. Isto significa que todo programa *deve* ter um main em algum lugar. main normalmente invocará outras funções para desenvolver seu trabalho, algumas do próprio programa, outras de bibliotecas de funções previamente escritas.

Um método de comunicação de dados entre funções é por argumentos. Os parênteses seguindo o nome da função envolvem a lista de argumentos; aqui, main é uma função sem argumentos, indicado por (). As chaves { } envolvem os comandos que formam a função; elas são análogas ao DO-END do PL/1, ou o BEGIN-END do Algol, Pascal, e assim por diante. Uma função é ativada por seu nome, seguido de uma lista de argumentos entre parênteses. Não há comandos CALL como em Fortran ou em PL/1. Os parênteses devem estar presentes mesmo quando não há argumentos.

A linha

```
printf (“primeiro programa\n”);
```

é uma chamada de função, que chama a função printf, com o argumento “primeiro programa\n”. printf é uma função de biblioteca que imprime no terminal (a menos que outro destino seja especificado). Nesse caso ela imprime a cadeia de caracteres que compõem seu argumento.

Uma sequência de qualquer número de caracteres entre aspas “...” é chamada *cadeia de caracteres* ou *constante do tipo cadeia*. Por ora, nosso único uso de cadeias de caracteres será como argumento para printf e outras funções.

A sequência \n na cadeia é a notação C para o caractere *nova-linha*, que, quando impresso, provoca o avanço do cursor do terminal para o início da próxima linha. Se você omitir o \n (um experimento valioso), você encontrará sua saída não terminada por uma alimentação de linha. A única maneira de se obter um caractere de nova-linha no argumento de printf é com o uso de \n; se você tentar algo como

```
printf ("primeiro programa
");
```

o compilador C imprimirá um diagnóstico não amigável sobre a falta de aspas.

printf nunca fornece uma nova-linha automaticamente, de forma que múltiplas ativasções podem ser usadas para construir uma linha de saída, passo a passo. Nosso primeiro programa poderia ter sido escrito como

```
main ( )
{
    printf ("primeiro ");
    printf ("programa");
    printf ("\n");
}
```

para produzir uma saída idêntica.

Observe que \n representa somente um único caractere. Uma *sequência de escape* tal como \n provê um mecanismo geral e extensível para a representação de caracteres difíceis de obter ou invisíveis. Entre os outros que C provê, estão \t para tabulação, \b para retrocesso, \" para a aspa, e \\ para a contrabarra propriamente dita.

Exercício 1-2. Verifique o que acontece quando a cadeia argumento de printf contém \x, onde x é algum caractere não listado acima.

1.2 Variáveis e Aritmética

O próximo programa imprime a seguinte tabela de graus Fahrenheit e seus correspondentes graus centígrados ou Celsius, usando a fórmula $C = (5/9)(F - 32)$.

0	- 17.8
20	- 6.7
40	4.4
60	15.6
...	...
260	126.7
280	137.8
300	148.9

Aqui está o programa.

```
/* imprimir a tabela de conversao Fahrenheit-Celsius
   para f = 0, 20, ..., 300 */
main { }
{
```

```

int inicio, fim, incr;
float fahr, celsius;

inicio = 0;      /* limite inferior da tabela */
fim = 300;      /* limite superior */
incr = 20;      /* incremento */

fahr = inicio;
while (fahr <= fim) {
    celsius = (5.0/9.0) * (fahr - 32.0);
    printf ("%4.0f %6.1f\n", fahr, celsius);
    fahr = fahr + incr;
}
}

```

As primeiras duas linhas

```

/* imprimir a tabela de conversao Fahrenheit-Celsius
para f = 0, 20, ..., 300 */

```

são um *comentário* que neste caso explica brevemente o que o programa faz. Quaisquer caracteres entre `/*` e `*/` são ignorados pelo compilador; eles podem ser usados livremente para tornar o programa mais fácil de ser entendido. Comentários podem aparecer em qualquer lugar onde possa aparecer um espaço em branco ou uma nova-linha.

Em C, *todas* as variáveis devem ser declaradas antes de usadas, normalmente, no início da função, antes de qualquer comando executável. Se você esquecer uma declaração, obterá um diagnóstico do compilador. Uma declaração consiste de um *tipo* e de uma lista de variáveis que tenham este tipo, como em

```

int inicio, fim, incr;
float fahr, celsius;

```

O tipo `int` implica que as variáveis listadas são *inteiras*; `float` define *ponto flutuante*, isto é, números que podem ter uma parte fracionária. A precisão dos tipos `int` e `float` depende da máquina particular que você está usando. No PDP-11, por exemplo, um `int` é um número de 16 bits com sinal, isto é, um número que varia de -32768 a $+32767$. Um `float` é uma quantidade de 32 bits, que comporta aproximadamente 7 dígitos decimais significativos, com magnitude entre aproximadamente 10^{-38} e 10^{+38} . O Capítulo 2 fornece o tamanho para outras máquinas.

C provê vários outros tipos de dados básicos além de `int` e `float`:

<code>char</code>	caractere — um único byte
<code>short</code>	inteiro curto
<code>long</code>	inteiro longo
<code>double</code>	ponto flutuante em dupla precisão.

Os tamanhos desses objetos são também dependentes da máquina; detalhes estão no Capítulo 2. Há também *arranjos*, *estruturas* e *uniões* desses tipos básicos, *apontadores* para eles, e *funções* que os retornam; todos serão vistos ao longo do livro.

O cálculo atual no programa de conversão de temperatura começa com as atribuições

```
inicio = 0;
fim     = 300;
incr    = 20;
fahr    = inicio;
```

que atribuem às variáveis seus valores iniciais. Comandos individuais são terminados por ponto-e-vírgula.

Cada linha da tabela é computada da mesma forma, e usamos, portanto, um laço que se repete uma vez por linha; essa é a finalidade do comando `while`

```
while (fahr <= fim) {
    ...
}
```

A condição entre parênteses é testada. Se ela é verdadeira (`fahr` é menor ou igual a `fim`), o corpo do laço (todos os comandos entre `{` e `}`) é executado. Então a condição é retestada, e se verdadeira, o corpo é executado novamente. Quando o teste se tornar falso (`fahr` exceder `fim`) o laço terminará, e a execução continuará no comando que segue o laço. Não há mais comandos neste programa, e ele termina.

O corpo de um `while` pode ter um ou mais comandos entre chaves, como no conversor de temperatura, ou um único comando sem chaves como em

```
while (i < j)
    i = 2 * i;
```

Em ambos os casos, os comandos controlados pelo `while` estão endentados por uma posição de tabulação de forma que você possa ver rapidamente que comandos estão dentro do laço. A endentação enfatiza a estrutura lógica do programa. Embora C seja muito liberal quanto ao posicionamento de comandos, a endentação correta e o uso de espaços são críticos na construção de programas fáceis de serem entendidos pelas pessoas. Recomendamos escrever somente um comando por linha, e (normalmente) deixar espaços em volta dos operadores. A posição das chaves é menos importante; escolhemos um entre vários estilos populares. Escolha um estilo que lhe agrade, e use-o sempre.

A maior parte do trabalho é feita no corpo do laço. A temperatura Celsius é computada e atribuída a `celsius` pelo comando

```
celsius = (5.0/9.0) * (fahr - 32.0);
```

A razão para usarmos `5.0/9.0` ao invés da construção mais simples `5/9` é que, em C, como na maioria das outras linguagens, a divisão inteira *trunca* o resultado, de forma que qualquer parte fracionária é perdida. Então `5/9` é zero e, é claro, assim seriam também todas as temperaturas. Um ponto decimal em uma constante indica que ela é do tipo ponto flutuante, assim `5.0/9.0` representa `0.555...`, que é o que queremos.

Nós também escrevemos `32.0` ao invés de `32`, apesar de `fahr` ser do tipo `float`, e `32` ser convertido automaticamente para `float` (para `32.0`) antes da subtração. Por uma questão de estilo, é aconselhável escrever constantes de ponto flutuante com ponto decimal explícito mesmo que tenham valores inteiros; isso enfatiza sua natureza de ponto flutuante para os leitores, e assegura que o compilador verá as coisas ao seu modo também.

As regras detalhadas da conversão de inteiros para ponto flutuante estão no Capítulo 2. Por hora, observe que o comando

```
fahr = inicio;
```

e o teste

```
while (fahr <= fim)
```

funcionam como esperado – o int é convertido para float antes da operação ser feita.

Esse exemplo também mostra um pouco mais sobre a operação de printf. printf é na realidade uma função geral de conversão de formatos, que descreveremos por completo no Capítulo 7. Seu primeiro argumento é uma cadeia de caracteres a ser impressa, onde cada sinal % indica onde um dos outros argumentos deve ser substituído, e sob que formato será impresso. Por exemplo, no comando

```
printf ("%4.0f %6.1f\n", fahr, celsius);
```

a especificação %4.0f diz que um número de ponto flutuante deve ser impresso num campo de pelo menos 4 caracteres, sem dígitos após o ponto decimal. %6.1f descreve outro número que ocupa pelo menos 6 caracteres, com 1 dígito após o ponto decimal, análogo ao F6.1 do Fortran ou F(6,1) do PL/1. Partes das especificações podem ser omitidas: %6f diz que o número deve ocupar pelo menos 6 caracteres; %2f requer 2 posições após o ponto decimal, mas não define o tamanho (e %f simplesmente indica a impressão de um número de ponto flutuante. printf também reconhece %d para inteiro decimal, %o para octal, %x para hexadecimal, %c para caractere, %s para cadeia de caracteres, e %% para o próprio %.

Cada construção do tipo % no primeiro argumento de printf casa-se com o segundo (terceiro, etc.) argumento; eles devem, correspondentemente, se alinhar corretamente por número e tipo, caso contrário você obterá respostas sem significado.

A propósito, printf não faz parte da linguagem C; não há entrada ou saída definida na linguagem. Não há nenhuma mágica sobre printf; ela é uma função útil que faz parte da biblioteca padrão de rotinas que são normalmente acessíveis a um programa C. Para nos concentrarmos somente na linguagem C, não vamos falar muito sobre entrada e saída até o Capítulo 7. Em particular, vamos deixar o assunto de entrada formatada até lá. Se você tem de ler números, leia sobre a função scanf no Capítulo 7, seção 7.4. scanf é muito parecida com printf, exceto que ela lê a entrada ao invés de imprimir na saída.

Exercício 1-3. Modifique o programa de conversão de temperatura para imprimir um cabeçalho acima da tabela.

Exercício 1-4. Escreva um programa para imprimir a tabela correspondente de Celsius a Fahrenheit.

1.3 O Comando For

Como você poderia esperar, há uma infinidade de maneiras de se escrever um programa; vamos tentar uma variante do conversor de temperatura.

```
main ( ) /* tabela de conversão Fahrenheit-Celsius */  
{  
    int fahr;
```

```
for (fahr = 0; fahr <= 300; fahr = fahr + 20)
    printf ("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
```

```
}
```

Ele produz as mesmas respostas, mas certamente parece diferente. Uma mudança principal é a eliminação da maioria das variáveis; somente `fahr` permaneceu, como `int` (para mostrar a conversão `%d` em `printf`). Os limites inferior e superior e o tamanho do incremento aparecem como constantes no comando `for`, que é uma nova construção, e a expressão que calcula a temperatura Celsius aparece agora como o terceiro argumento de `printf`, ao invés de ser um comando separado de atribuição.

Esta última mudança é um exemplo de uma regra geral em C – em qualquer lugar onde é permitido o uso de uma variável de algum tipo, você pode usar uma expressão deste tipo. Visto que o terceiro argumento de `printf` deve ser um valor de ponto flutuante para casar com a especificação `%6.1f`, qualquer expressão do tipo ponto flutuante pode aparecer aí.

O `for` é um laço, uma generalização do `while`. Se você compará-lo com o primeiro `while` visto, sua operação deverá ser clara. Ele contém três partes, separadas por ponto-e-vírgula. A primeira parte

```
fahr = 0
```

é executada uma vez, antes do laço ser iniciado. A segunda parte é o teste ou condição que controla o laço:

```
fahr <= 300
```

Esta condição é avaliada; se verdadeira, o corpo do laço (um único comando `printf`) é executado. Finalmente, o passo de reinicialização

```
fahr = fahr + 20
```

é feito, e a condição é reavaliada. O laço termina quando a condição se torna falsa. Como no `while`, o corpo do laço pode ser um único comando ou um grupo de comandos entre chaves. A inicialização e a reinicialização podem ser qualquer expressão simples.

A escolha entre `while` e `for` é arbitrária, baseada no que parece mais claro para o usuário. O `for` é normalmente apropriado para laços onde a inicialização e a reinicialização são comandos simples e logicamente relacionados, pois é mais compacto que o `while` e mantém os comandos de controle do laço juntos no mesmo local.

Exercício 1-5. Modifique o programa de conversão de temperatura para imprimir a tabela em ordem inversa, isto é, de 300 graus até 0 grau.

1.4 Constantes Simbólicas

Uma última observação antes de deixarmos (para sempre) o conversor de temperatura. Não é boa prática de programação utilizar “números mágicos” como 300 e 20 num programa; eles trazem pouca informação para alguém que tenha de ler o programa posteriormente, e são difíceis de serem alterados de forma sistemática. Felizmente, C provê uma forma de evitar números mágicos. Com a construção `define` no início de um programa, você pode definir um *nome simbólico* ou uma *constante simbólica* como sendo uma cadeia de caracteres particular. Desta forma, o compilador troca toda ocorrência (não contida entre aspas) do nome simbólico pela cadeia correspondente. A substituição do nome pode ser, na realidade, qualquer texto, não apenas números.

```
#define INICIO 0    /* limite inferior da tabela */
#define FIM 300    /* limite superior */
#define INCR 20    /* incremento */

main ( ) /* Conversao Fahrenheit-Celsius */
{
    int fahr;

    for (fahr = INICIO; fahr <= FIM; fahr = fahr + INCR)
        printf ("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
}
```

As quantidades INICIO, FIM e INCR são constantes, e, por isso, elas não aparecem em declarações. Nomes simbólicos são comumente escritos em letras maiúsculas para serem facilmente diferenciados de nomes de variáveis em letras minúsculas. Observe que não há ponto-e-vírgula ao fim da definição. Desde que toda a linha após o nome definido é substituída, haveria pontos-e-vírgulas em excesso no for.

1.5 Uma Coleção de Programas Úteis

Vamos considerar agora uma família de programas que façam operações simples sobre caracteres. Você descobrirá que muitos programas são versões expandidas dos protótipos aqui discutidos.

Entrada e Saída de Caracteres

A biblioteca padrão provê funções para a leitura e escrita de um caractere por vez. `getchar ( )` obtém o *próximo caractere de entrada* cada vez que é chamada, e retorna o caractere como seu valor. Isto é, após

```
c = getchar ( )
```

a variável `c` contém o próximo caractere da entrada. Os caracteres normalmente vêm do terminal, mas essa restrição não nos interessará até o Capítulo 7.

A função `putchar (c)` é o complemento de `getchar`:

```
putchar (c)
```

imprime o conteúdo da variável `c` em algum meio de saída, normalmente o terminal. Chamadas a `putchar` e `printf` podem ser intercaladas; a saída aparecerá na ordem das chamadas.

Como no caso de `printf`, não há nada de especial com `getchar` e `putchar`. Elas não são partes integrantes de C, mas estão disponíveis universalmente.

Cópia de Arquivos

Dados `getchar` e `putchar`, você pode escrever uma quantidade surpreendente de programas úteis sem conhecer mais nada sobre entrada e saída. O exemplo mais simples é um programa que copia sua entrada na sua saída, caractere a caractere. Em linhas gerais, temos:

```
leia um caractere
enquanto (caractere não é sinalizador fim de arquivo)
    imprima caractere lido
    leia novo caractere
```

Convertendo em C, temos:

```
main ( )    /* copia entrada na saída: versao 1 */
{
    int c;
    c = getchar ( ) ;
    while (c != EOF) {
        putchar (c) ;
        c = getchar ( ) ;
    }
}
```

O operador relacional `!=` significa “não igual a”.

O problema principal é detectar o fim da entrada. Por convenção, `getchar` retorna um valor que não é um caractere válido quando encontra o fim da entrada; desta forma, os programas podem determinar quando não há mais dados na entrada. A única complicação é que há duas convenções comumente usadas para o valor do fim de arquivo. Evitamos este problema por enquanto usando o nome simbólico `EOF` para representar este valor, sempre que possível. Na prática, `EOF` será ou `-1` ou `0`, de forma que o programa deve ser precedido pela definição apropriada:

```
#define EOF -1
```

ou

```
#define EOF 0
```

para funcionar corretamente. Pelo uso de uma constante simbólica `EOF` para representar o valor que `getchar` retorna ao fim do arquivo, garantimos que somente uma coisa (`EOF`), no programa dependa de um valor numérico específico.

Também declaramos `c` como `int` e não `char`, de modo que ele possa conter o valor que `getchar` retorna. Como poderemos ver no Capítulo 2, este valor é na realidade do tipo `int`, pois deve ser capaz de representar `EOF` além de todos os possíveis `char`'s.

O programa para copiar poderia ser escrito mais resumidamente por um programador C experiente. Em C, qualquer atribuição, tal como:

```
c = getchar ( )
```

pode ser usada numa expressão; seu valor é simplesmente o valor atribuído à parte esquerda da atribuição. Se a atribuição de um caractere a `c` é posta dentro da parte de teste do `while`, o programa de cópia de arquivo poderia ser escrito como

```
main ( )    /* copia entrada na saída: versao 2 */
{
    int c;

    while ((c = getchar ( )) != EOF)
        putchar (c) ;
}
```

O programa lê um caractere, o atribui a `c`, e então testa se o caractere é o fim de arquivo. Caso negativo, o corpo do `while` é executado, imprimindo o caractere. O `while` então se repete. Quando o fim da entrada é encontrado, o `while` termina, e, conseqüentemente `main`.

Essa versão centraliza a entrada — há somente uma chamada a `getchar` — e encolhe o programa. O aninhamento de uma atribuição num teste é uma parte de C que possibilita uma concisão valiosa (possibilita também a criação de programas impenetráveis, tendência essa que tentaremos coibir).

É importante reconhecer que os parênteses circundando a atribuição são realmente necessários. A precedência de `!` é maior que a de `=`, isto significa que na falta de parênteses, o teste relacional `! =` seria feito antes da atribuição = Assim, o comando:

```
c = getchar ( ) ! = EOF
```

é equivalente a

```
c = (getchar ( ) ! = EOF)
```

que tem o efeito indesejável de atribuir 1 (verdadeiro) ou 0 (falso) a `c`, dependendo se a chamada encontrou ou não o fim de arquivo.

Contagem de Caracteres

O próximo programa conta caracteres; ele é uma pequena elaboração do programa para copiar arquivos.

```
main ( )      /* conta caracteres na entrada */
{
    long nc;

    nc = 0;
    while (getchar ( ) ! = EOF)
        ++ nc;
    printf ("%ld\n", nc);
}
```

O comando

```
++ nc;
```

mostra um novo operador, `++`, que significa *incremente de um*. Você poderia escrever `nc = nc + 1`, mas `++ nc` é mais conciso e freqüentemente mais eficiente. Há um operador correspondente — para decrementar de 1. Os operadores `++` e `--` podem estar prefixados (`++ nc`) ou pós-fixados (`nc ++`); essas duas formas têm valores diferentes em expressões, como veremos no Capítulo 2, mas ambos `++ nc` e `nc ++` incrementam `nc`. Por ora usaremos o incremento prefixado.

O programa de contagem de caracteres acumula seu contador numa variável do tipo `long` ao invés de `int`. No PDP-11 o valor máximo de um `int` é 32767, e seriam necessários poucos caracteres para estourar o contador se ele fosse declarado como `int`; no Honeywell e IBM, `long` e `int` são sinônimos e são muito maiores. A especificação de conversão `%ld` indica ao `printf` que o argumento correspondente é um inteiro do tipo `long`.

Para enfrentar números ainda maiores, você pode usar um `double` (ponto flutuante de dupla precisão). Usaremos também um comando `for` ao invés de `while`, para ilustrar uma forma alternativa de escrever o laço.

```

main ( ) /* conta caracteres na entrada */
{
    double nc;

    for (nc = 0; getchar ( ) != EOF; ++ nc)
        ;
    printf ("%f\n", nc) ;
}

```

printf usa %f para float e double; %.Of suprime a impressão da parte fracionária não existente.

O corpo do laço for é *vazio* aqui, porque todo o trabalho é feito nas partes de teste e re-inicialização. Entretanto, a gramática de C requer um corpo para o comando for. O ponto-e-vírgula isolado, definido tecnicamente como um *comando nulo*, é colocado para atender à gramática. Nós o colocamos numa linha separada por questão de legibilidade.

Antes de deixarmos o programa de contagem de caracteres, observemos que, se a entrada não contém nenhum caractere, o teste do while ou for falha no primeiro teste da chamada a getchar, com o programa dando como resposta o valor 0, que é o correto. Essa é uma observação importante. Uma das coisas mais agradáveis do while e for é o fato de ambos fazerem o teste no *início* do laço, antes de prosseguir no corpo. Se não há nada para fazer, nada é feito, mesmo que o corpo do laço não seja executado. Programas devem agir inteligentemente quando manipulam entradas vazias. Os comandos while e for ajudam a garantir que coisas razoáveis sejam feitas em condições limites.

Contagem de Linhas

O próximo programa conta *linhas* na entrada. Linhas de entrada devem terminar com o caractere de nova-linha \n que tem sido acrescentado religiosamente ao fim de toda linha gravada.

```

main ( ) /* conta linhas na entrada */
{
    int c, nl ;

    nl = 0 ;
    while ((c = getchar ( )) != EOF)
        if (c == '\n')
            ++ nl ;
    printf ("%d\n", nl) ;
}

```

O corpo do while consiste agora de um if que controla o incremento ++ nl. O comando if testa a condição entre parênteses e, se ela for verdadeira, executa o comando (ou grupo de comandos entre chaves) que segue. Usamos endentação novamente para mostrar “o que” é controlado “por quem”.

O duplo sinal de igualdade == é a notação C para “é igual a” (como o .EQ. do Fortran). Este símbolo é usado para diferenciar o teste de igualdade do = simples usado para atribuição. Como a atribuição é aproximadamente duas vezes mais freqüente que o teste de igualdade em programas típicos, é apropriado que aquela use um símbolo da metade do comprimento.

Quando um caractere é escrito entre apóstrofes, produz um valor igual, numericamente, ao código do caractere na máquina; isto é chamado de *constante do tipo caractere*. Por exemplo, 'A' é uma constante do tipo caractere; no conjunto de caracteres ASCII seu valor é 65, a representação interna do caractere A. É claro que 'A' é preferível a 65: seu significado é óbvio, e é independente de um conjunto particular de caracteres.

As seqüências de escape usadas em cadeias de caracteres são legais, também, em constantes do tipo caractere, de forma que, em testes e expressões aritméticas, '\n' representa o valor do caractere de nova-linha. Você deve observar que '\n' é um único caractere, e em expressões, é equivalente a um único inteiro; por outro lado, "\n" é uma cadeia de caracteres que contém somente um caractere. O tópico de cadeias versus caracteres é discutido mais amiúde no Capítulo 2.

Exercício 1-6. Escreva um programa que conte espaços, caracteres de tabulação e de nova-linha.

Exercício 1-7. Escreva um programa que copie sua entrada na saída, trocando cada cadeia de dois ou mais espaços por um único espaço.

Exercício 1-8. Escreva um programa que troque cada caractere de tabulação da entrada pela seqüência >, retrocesso, —, na saída, produzindo >, e cada retrocesso na entrada pela seqüência similar <, tornando os caracteres tabulação e de retrocesso da entrada visíveis na saída.

Contagem de Palavras

O quarto de nossa série de programas úteis conta linhas, palavras, e caracteres, com a definição vaga de palavra sendo uma seqüência de caracteres sem espaço, caractere de tabulação ou de nova-linha (esta é uma versão simplória do utilitário *wc* do UNIX.).

```
#define SIM 1
#define NAO 0

main ( ) /* conta linhas, palavras, carac. na entrada */
{
    int c, nl, np, nc, empalavra;

    empalavra = NAO;
    nl = np = nc = 0;
    while ((c = getchar ( )) != EOF) {
        ++ nc;
        if (c == '\n')
            ++ nl;
        if (c == ' ' || c == '\n' || c == '\t')
            empalavra = NAO;
        else if (empalavra == NAO) {
            empalavra = SIM;
            ++ np;
        }
    }
}
```



```
    printf ("%d %d %d\n", nl, np, nc) ;
}
```

Cada vez que o programa encontra o primeiro caractere de uma palavra, ele o conta. A variável *empalavra* registra se o programa está no momento dentro de uma palavra ou não; inicialmente ele não está dentro de uma palavra, o que é assinalado pelo valor NAO. Preferimos as constantes simbólicas SIM e NAO aos literais 1 e 0 porque elas tornam o programa mais legível. É claro que, num programa tão pequeno quanto esse, isso faz pouca diferença, mas em programas grandes, o ganho em clareza compensa o esforço extra para escrevê-los inicialmente dessa forma. Você verá também que é mais fácil fazer mudanças em programas onde números aparecem somente como constantes simbólicas.

A linha

```
nl = np = nc = 0 ;
```

atribui às três variáveis o valor zero. Isto não é um caso especial, mas a consequência do fato de que uma atribuição tem um valor e é associativa da direita para a esquerda. Poderíamos ter escrito

```
nc = (nl = (np = 0)) ;
```

O operador `||` significa OU, de forma que a linha

```
if (c == ' ' || c == '\n' || c == '\t')
```

significa “se *c* é um espaço *ou* *c* é uma nova-linha *ou* *c* é uma tabulação . . . ” (a sequência de escape `\t` é uma representação visível do caractere de tabulação). Há um operador correspondente `&&` para E. Expressões conectadas por `&&` ou `||` são avaliadas da esquerda para direita, e é garantido que a avaliação parará assim que a veracidade ou falsidade for conhecida. Então, se *c* contém um espaço em branco, não é necessário testar se ele contém um caractere de nova-linha ou de tabulação, de forma que esses testes *não* são feitos. Isso não é muito importante aqui mas em situações mais complicadas, como logo veremos.

O exemplo também mostra o comando `else` do C, que especifica uma ação alternativa para ser feita se a condição do comando `if` for falsa. A forma geral é

```
if (expressão)
    comando-1
else
    comando-2
```

Um e somente um dos dois comandos associados com o `if-else` é executado. Se a *expressão* é verdadeira, *comando-1* é executado; se não, *comando-2* é executado. Cada *comando* pode, de fato, ser mais complicado. No programa de contagem de palavras, o único comando após o `else` é um `if` que controla dois comandos entre chaves.

Exercício 1-9. Como você poderia testar o programa de contagem de palavras? Quais seriam alguns casos limites?

Exercício 1-10. Escreva um programa que imprima palavras da sua entrada, uma por linha.

Exercício 1-11. Revise o programa de contagem de palavras para usar uma definição melhor de “palavra”, por exemplo, uma sequência de letras, dígitos e apóstrofes que comece com uma letra.

1.6 Arranjos

Vamos escrever um programa para contar o número de ocorrências de cada dígito, espaço em branco, (espaços, caracteres de tabulação e de nova-linha) e todos os outros caracteres. Isto é artificial, é claro, mas nos permite ilustrar vários aspectos de C num único programa.

Há doze categorias de entrada, de forma que é conveniente usar um arranjo para contar o número de ocorrências de cada dígito, ao invés de dez variáveis individuais. Aqui está uma versão do programa:

```
main ( )      /* conta dígitos, espaço branco, outros */
{
    int c, i, nbranco, noutro;
    int ndigito[10];

    nbranco = noutro = 0;
    for (i = 0; i < 10; ++i)
        ndigito[i] = 0;

    while ( (c = getchar ( )) != EOF)
        if (c >= '0' && c <= '9')
            ++ ndigito [c - '0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++ nbranco;
        else
            ++ noutro;

    printf ("dígitos =");
    for (i = 0; i < 10; ++i)
        printf (" %d", ndigito[i]);
    printf ("\nespaço branco = %d, outro = %d\n",
            nbranco, noutro);
}
```

A declaração

```
int ndigito[10];
```

declara `ndigito` como sendo um arranjo de 10 inteiros. Índices de arranjos sempre começam de 0 em C (ao invés de 1 como em Fortran ou PL/I), de forma que os elementos são `ndigito[0]`, `ndigito[1]`, ..., `ndigito[9]`. Isso se reflete nos laços `for` que inicializam e imprimem o arranjo.

Um índice pode ser qualquer expressão inteira, incluindo, é claro, variáveis inteiras tais como `i`, e constantes inteiras.

Este programa particular utiliza várias propriedades da representação de dígitos como caracteres. Por exemplo, o teste

```
if (c >= '0' && c <= '9') ...
```

determina se o caractere em `c` é um dígito. Se `for`, o valor numérico do mesmo é

```
c - '0'
```

Isto funciona somente se '0', '1', etc., são positivos, em ordem crescente, e se não há nada além de dígitos entre '0' e '9'. Felizmente, isto é verdadeiro para todos os conjuntos convencionais de caracteres.

Por definição, a aritmética envolvendo char's e int's converte tudo para int antes de prosseguir, de forma que variáveis e constantes do tipo caractere são essencialmente idênticas a int's em contextos aritméticos. Isto é muito natural e conveniente; por exemplo, `c - '0'` é uma expressão inteira com valor entre 0 e 9 correspondente ao caractere '0' a '9' armazenado em `c`, e é então um subíndice válido para o arranjo `ndigito`.

A decisão que verifica se um caractere é um dígito, um espaço em branco ou alguma outra coisa é feita com a sequência

```
if (c >= '0' && c <= '9')
    ++ ndigito [c - '0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++ nbranco;
else
    ++ noutro;
```

A sequência

```
if (condicao)
    comando
else if (condicao)
    comando
else
    comando
```

ocorre frequentemente em programas como uma forma de expressar decisões múltiplas. O código é simplesmente lido do topo até que alguma condição seja satisfeita; neste ponto, o comando correspondente é executado, e a construção inteira termina. (É claro que comando pode ser vários comandos entre chaves.) Se nenhuma das condições é satisfeita, o comando após o `else` final é executado, quando presente. Se o `else` e o comando finais são omitidos (como no programa de contagem de palavras), nenhuma ação é feita. Pode haver um número arbitrário de construções do tipo

```
else if (condicao)
    comando
```

entre o `if` inicial e o `else` final. Como forma de estilo, é aconselhável formatar essa construção como indicado aqui, para que longas decisões não esbarrem na margem direita da página.

O comando `switch`, a ser discutido no Capítulo 3, provê uma outra forma de escrever um desvio múltiplo que é particularmente útil quando a condição a ser testada é a de ver se alguma expressão do tipo inteiro ou caractere casa com um elemento de um conjunto de valores constantes. Para fins de comparação, apresentaremos uma versão deste programa usando um `switch` no Capítulo 3.

Exercício 1-12. Escreva um programa que imprima um histograma do tamanho das palavras da entrada. É mais fácil desenhar o histograma na horizontal; uma representação vertical é mais desafiadora.

1.7 Funções

Em C, uma *função* é equivalente a uma sub-rotina ou função em Fortran, ou um procedimento em Pascal, PL/1, etc. Uma função fornece um meio conveniente de encapsular alguma computação em uma caixa preta, que pode ser usada sem preocupação quanto a seus detalhes internos. Funções são a única maneira de lidarmos com a complexidade potencial de grandes programas. Com funções projetadas apropriadamente, é possível ignorar *como* um trabalho é feito; conhecer *o que* é feito é suficiente. C é projetada para tornar o uso de funções fácil, conveniente e eficiente; você vai ver, freqüentemente, uma função com poucas linhas e ativada somente uma vez, só porque ela esclarece alguma parte do código.

Até agora nós usamos funções como `printf`, `getchar` e `putchar` que nos foram fornecidas; agora é hora de escrevermos algumas nós mesmos. Desde que C não tem operador de exponenciação como o `**` do Fortran ou PL/1, vamos ilustrar o mecanismo de definição de função escrevendo uma função `pot (m, n)` que eleva um inteiro `m` à potência positiva inteira `n`. Isto é, o valor de `pot (2, 5)` é 32. Esta função, certamente, não faz todo o trabalho de `**` já que só manipula potências positivas de pequenos inteiros, mas é melhor confundir um só conceito por vez. Aqui está a função `pot` e o programa principal que a chama, de forma que você possa ver a estrutura inteira.

```
main ( )      /* teste da funcao de potencia */
{
    int i;

    for (i = 0; i < 10; ++ i)
        printf ("%d %d %d\n", i, pot (2, i), pot (- 3, i) );
}

pot (x, n)     /* eleva x a potencia n; n > 0 */
int x, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++ i)
        p = p * x;
    return (p);
}
```

Cada função tem a forma:

```
nome (lista de argumentos, se houver)
declaracoes de argumentos, se houver
{
    declaracoes
    comandos
}
```

As funções podem aparecer em qualquer ordem, e em um ou dois arquivos-fonte. É claro que, se o fonte aparece em dois arquivos, você vai ter que dizer mais coisas para compilá-lo e carregá-lo do que se ele aparecesse em um único arquivo, mas isso diz respeito a siste-

mas operacionais e não é um atributo da linguagem. Por ora, presumiremos que ambas as funções estão no mesmo arquivo de forma que o que você aprendeu para rodar um programa em C não tenha que mudar. A função `pot` é chamada duas vezes na linha

```
printf ("%d %d %d\n", i, pot (2, i), pot (- 3, i) );
```

Cada chamada passa dois argumentos a `pot`, a qual retorna a cada vez, um inteiro para ser formatado e impresso. Numa expressão, `pot (2, i)` é um inteiro assim como 2 e `i` o são. (Nem toda função produz um valor inteiro; veremos isso no Capítulo 4.)

Em `pot` os argumentos devem ser declarados apropriadamente de forma que seus tipos sejam conhecidos. Isto é feito pela linha

```
int x, n;
```

em seguida ao nome da função. A declaração de argumentos fica entre a lista de argumentos e o abre-chave; cada declaração termina por um ponto-e-vírgula. Os nomes usados por `pot` para seus argumentos são puramente *locais* a `pot`, e não são acessíveis a qualquer outra função: outras rotinas podem usar os mesmos nomes sem conflito. Isto é válido também para as variáveis `i` e `p`: o `i` em `pot` não está relacionado com o `i` em `main`.

O valor que `pot` calcula é retornado para `main` pelo comando `return`, como em PL/1. Qualquer expressão pode ocorrer entre os parênteses. Uma função não precisa retornar um valor; um comando `return` sem expressão faz com que o controle, sem nenhum valor útil, retorne a quem o chama, como se houvesse encontrado o fim da função no fechamento final.

Exercício 1-13. Escreva um programa que converta sua entrada em letras minúsculas, usando uma função `minuscule` (`c`) que retorne `c` se `c` não é uma letra, e a minúscula correspondente a `c` se é uma letra.

1.8 Argumentos — Chamada por Valor

Um aspecto de funções C pode não ser familiar a programadores que usem outras linguagens, particularmente Fortran e PL/1. Em C, todos os argumentos são passados “por valor”. Isto significa que, à função chamada, é dada uma cópia dos valores dos argumentos em variáveis temporárias (na realidade numa pilha) ao invés de seus endereços. Isso leva a algumas propriedades diferentes das vistas em linguagens tais como Fortran e PL/1 que usam “chamada por referência”, onde a rotina chamada recebe o endereço do argumento, e não seu valor.

A diferença principal é que, em C, uma função chamada *não pode* alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária.

Chamada por valor é uma vantagem, entretanto, e não uma desvantagem. Leva normalmente a programas mais compactos com poucas variáveis adicionais, porque os argumentos podem ser tratados como variáveis locais convenientemente inicializadas na rotina chamada. Por exemplo, segue uma versão de `pot` que faz uso deste fato.

```
pot (x, n)    /* eleva x a potencia n; n > 0; versao 2 */
int x, n;
{
    int p;
```

```

for (p = 1; n > 0; -- n)
    p = p * x;
return (p);
}

```

O argumento *n* é usado como variável temporária, e é decrementado até zero; a variável *i* não é mais necessária. Tudo o que for feito com *n* dentro de *pot* não tem efeito no argumento que *pot* recebeu originalmente.

Quando necessário, é possível permitir a uma função modificar uma variável na rotina que chama. Quem chama deve fornecer o *endereço* da variável (tecnicamente, um *apontador* para a variável), e a função chamada deve declarar que o argumento é um apontador e deve referenciar indiretamente o valor atual da variável. Veremos isso em detalhes no Capítulo 5.

Quando o nome de um arranjo é usado como argumento, o valor passado para a função é o endereço do início do arranjo (não há cópia de elementos do arranjo). Pelo uso deste valor com índices, a função pode acessar e alterar qualquer elemento do arranjo. Este é o tópico da próxima seção.

1.9 Arranjos de Caracteres

Provavelmente, o tipo mais comum de arranjo em C é o de caracteres. Para ilustrar o uso de arranjos de caracteres, e funções que os manipulam, vamos escrever um programa que lê um conjunto de linhas e imprime a maior delas. O algoritmo básico é simples:

```

while (há outra linha)
    if (ela é maior que a anterior)
        salve-a junto com seu tamanho
imprima a maior linha

```

Este algoritmo deixa claro que o programa se divide naturalmente em partes. Uma parte obtém uma nova linha, outra testa-a, outra salva-a e o resto controla o processo.

Visto que as coisas se dividem tão facilmente, será melhor escrevê-las desta forma também. Vamos primeiro escrever uma função separada chamada *lelinha* que busca a *próxima linha* da entrada; é uma generalização de *getchar*. Para que a função seja útil em outros contextos, tentaremos fazê-la tão flexível quanto possível. No mínimo, *lelinha* deve sinalizar o fim de arquivo; um projeto mais geral e útil seria retornar o tamanho da linha, ou zero se o fim do arquivo foi encontrado. Zero nunca é um tamanho de linha válido, desde que toda linha tem ao menos um caractere — o de nova-linha.

Quando encontramos uma linha que é maior que a maior linha anterior, ela deve ser guardada em algum lugar. Isso sugere uma segunda função, *copia*, para copiar a nova linha maior para um lugar apropriado.

Finalmente, precisamos de um programa principal para controlar *lelinha* e *copia*. Eis aqui o resultado.

```

#define    MAXLINHA 1000 /* tamanho maximo da linha */

main ( )    /* acha a linha mais longa */
{
    int tam; /* tamanho corrente da linha */
    int max; /* tamanho maximo visto ate agora */
    char linha [MAXLINHA]; /* linha corrente */
    char guarda [MAXLINHA]; /* maior linha guardada */

    max = 0;
    while ((tam = lelinha (linha, MAXLINHA)) > 0)
        if (tam > max) {
            max = tam;
            copia (linha, guarda);
        }
    if (max > 0) /* entrada tinha uma linha */
        printf ("%s", guarda);
}

lelinha (s, lim) /* le a linha em s, retorna o tamanho */
char s[];
int lim;
{
    int c, i;

    for (i = 0; i < lim - 1 && (c = getchar ( )) != EOF && c != '\n'; ++ i)
        s [i] = c;
    if (c == '\n') {
        s [i] = c;
        ++ i;
    }
    s [i] = '\0';
    return (i);
}

copia (s1, s2) /* copia s1 em s2; assumá que cabe em s2 */
char s1 [ ], s2 [ ];
{
    int i;

    i = 0;
    while ((s2 [i] = s1 [i]) != '\0')
        ++ i;
}

```

main e lelinha se comunicam através de um par de argumentos e um valor retornado. Em lelinha, os argumentos são declarados pelas linhas

```
char s [ ];  
int lim;
```

que especificam que o primeiro argumento é um arranjo, e o segundo é um inteiro. O tamanho do arranjo `s` não é especificado em `lelinha` desde que ele é determinado em `main`. `lelinha` usa `return` para devolver um valor ao chamador, como no caso da função `pot`. Algumas funções retornam um valor útil; outras, tais como `copia`, são somente usadas por seu efeito, e não retornam nada.

`lelinha` coloca o caractere `\0` (o *caractere nulo*, cujo valor é zero) no fim do arranjo, para marcar o fim da cadeia de caracteres. Essa convenção é também usada pelo compilador C: quando uma constante do tipo cadeia tal como

`"dia\n"`

aparece num programa C, o compilador cria um arranjo de caracteres contendo os caracteres da cadeia, e a encerra com um `\0` de modo que funções tais como `printf` possam detectar o fim da cadeia:

d	i	a	\n	\0
---	---	---	----	----

O formato `%s` em `printf` espera uma cadeia representada nesse formato. Se você examinar `copia`, você descobrirá que ela também se baseia no fato de que seu argumento de entrada `s1` termina com `\0`, e ela copia este caractere no seu argumento de saída `s2`. (Tudo isso implica que o texto normal não contém o caractere `\0`.)

Vale mencionar que, mesmo um programa tão pequeno quanto este, apresenta alguns problemas de projeto. Por exemplo, o que deve `main` fazer se for encontrada uma linha maior que o limite? `lelinha` funciona apropriadamente, pois ela interrompe a coleta de caracteres quando o arranjo está cheio, mesmo se o caractere de nova-linha não tiver sido encontrado. Pelo teste do tamanho e do último caractere retornado, `main` pode determinar se a linha é muito grande, e então prosseguir da forma desejada. Para abreviar o programa, ignoramos este caso.

Um usuário de `lelinha` não tem como saber, antecipadamente, o tamanho que uma linha de entrada pode ter; portanto `lelinha` verifica o estouro. Por outro lado, o usuário de `copia` já conhece (ou pode conhecer) o tamanho das cadeias, de modo que escolhemos não incluir a verificação de erro à função.

Exercício 1-14. Revise a rotina `main` do programa que imprime a maior linha para que ele imprima corretamente o tamanho da maior linha de entrada encontrada, e a maior quantidade de texto possível da mesma.

Exercício 1-15. Escreva um programa que imprima todas as linhas de entrada com mais de 80 caracteres.

Exercício 1-16. Escreva um programa que remova espaços em branco e caracteres de tabulação finais de cada linha de entrada e que delete as linhas inteiramente em branco.

Exercício 1-17. Escreva uma função `inverte(s)` que inverta a ordem dos caracteres da cadeia `s`. Use-a para escrever um programa que inverta sua entrada, linha a linha.

1.10 Escopo; Variáveis Externas

As variáveis em main (linha, guarda, etc.) são privativas ou locais a main; visto que elas são declaradas em main, nenhuma outra função tem acesso direto a elas. O mesmo é verdadeiro para variáveis de outras funções; por exemplo, a variável `i` em `lelinha` não tem relação alguma com `i` em `copia`. Cada variável local numa rotina passa a existir somente quando a função é chamada, e *deixa* de existir quando a função termina. Por essa razão, tais variáveis são conhecidas normalmente como *automáticas*, seguindo a terminologia de outras linguagens. Usaremos o termo *automática* daqui para frente para nos referir a essas variáveis locais dinâmicas. (O Capítulo 4 discute a classe de armazenamento *static*, onde variáveis locais retêm seus valores entre ativações da função.)

Devido às variáveis automáticas virem e irem com a ativação da função, elas não retêm seus valores de uma chamada para outra, e devem ser explicitamente inicializadas a cada ativação. Se elas não forem inicializadas, conterão lixo.

Como uma alternativa para variáveis automáticas, é possível definir variáveis que sejam *externas* a todas as funções, isto é, variáveis globais que possam ser acessadas pelo nome por qualquer função que assim o desejar. (Este mecanismo é semelhante ao `COMMON` do Fortran ou `EXTERNAL` do PL/1.) Visto que as variáveis externas sejam globalmente acessíveis, podem ser usadas, ao invés de listas de argumentos, para comunicar dados entre funções. Além do mais, devido às variáveis externas existirem permanentemente, ao invés de aparecerem com a ativação e desativação de funções, elas retêm seus valores mesmo quando as funções que as acessam deixam de existir.

Uma variável externa deve ser *definida* fora de qualquer função; isso aloca armazenamento para as mesmas. A variável deve também ser *declarada* em cada função que desejar acessá-la; isso pode ser feito por uma declaração explícita `extern` ou implicitamente pelo contexto. Para tornarmos a discussão concreta, vamos reescrever o programa que imprima a maior linha com linha, guarda e max sendo variáveis externas. Isso requer mudanças nas chamadas, declarações, e corpos de todas as três funções.

```
#define MAXLINHA 1000 /* tamanho maximo da linha */
```

```
char linha[MAXLINHA]; /* linha de entrada */  
char guarda[MAXLINHA]; /* linha mais longa */  
int max; /* tamanho maximo visto ate agora */
```

```
main ( ) /* acha a linha mais longa; versao especializada */  
{
```

```
    int tam;  
    extern int max;  
    extern char guarda [ ];
```

```
    max = 0;  
    while ((tam = lelinha ( )) > 0)  
        if (tam > max) {  
            max = tam;  
            copia ( );  
        }
```

```
    if (max > 0) /* entrada tinha uma linha */  
        printf ("%s", guarda);
```

```

    }

    lelinha ( ) /* versao especializada */
    {
        int c, i;
        extern char linha [ ];

        for (i = 0; i < MAXLINHA - 1
            && (c = getchar ( )) != EOF && c != '\n'; ++ i)
            linha [i] = c;
        if (c == '\n') {
            linha [i] = c;
            ++ i;
        }
        linha [i] = '\0';
        return (i);
    }

    copia ( ) /* versao especializada */
    {
        int i;
        extern char linha [ ], guarda [ ];

        i = 0;
        while ((guarda [i] = linha [i]) != '\0')
            ++ i;
    }

```

As variáveis externas em main, lelinha e copia são *definidas* pelas primeiras linhas do exemplo acima, que definem seus tipos e alocam área de armazenamento para as mesmas. Sintaticamente definições externas são iguais às declarações que já usamos anteriormente, mas, como ocorrem fora de funções, as variáveis são externas. Antes de uma função usar uma variável externa, o nome da mesma deve ser conhecido pela função. Uma maneira de fazer isso é escrever uma *declaração* extern na função; a declaração é igual à anterior, exceto que é precedida da palavra-chave extern.

Em certas circunstâncias, a declaração extern pode ser omitida. Se a definição externa de uma variável ocorrer no arquivo-fonte *antes* de seu uso numa função particular, então não há a necessidade de uma declaração extern na função. As declarações extern em main, lelinha e copia são, portanto, redundantes. De fato, a prática comum é colocar a definição de todas as variáveis externas no início do arquivo-fonte e, então, omitir todas as declarações extern.

Se o programa está em vários arquivos-fonte, e uma variável é definida em, digamos, *arquivo1* e usada em *arquivo2*, então uma declaração extern é necessária no *arquivo2* para conectar as duas ocorrências da variável. Este tópico é discutido mais detalhadamente no Capítulo 4.

Você deve observar que estamos usando as palavras *declaração* e *definição* cuidadosamente quando nos referimos a variáveis externas nessa seção. “Definição” refere-se ao local onde a variável está sendo criada ou à qual é atribuída uma área de armazenamento; “declara-

ção" refere-se ao local onde a natureza da variável é dada, sem alocação de área de armazenamento.

A propósito, há uma tendência de se fazer com que tudo em vista seja uma variável extern porque simplifica aparentemente a comunicação — listas de argumentos são curtas e as variáveis estão sempre onde você as quer. Mas variáveis externas estão sempre lá, mesmo quando você não as quer. Este estilo de codificação é cheio de perigos porque leva a programas cujas conexões de dados não são óbvias — variáveis podem ser mudadas de modo inesperado e inadvertidamente, e o programa é difícil de modificar se o for necessário. A segunda versão do programa da maior linha é inferior à primeira, em parte por essas razões, e em parte porque elimina a generalidade de duas funções bastante úteis, por tê-las amarrado às variáveis que elas manipulam.

Exercício 1-18. O teste do comando `for` na função `lelinha` é um tanto o quanto desajeitado. Reescreva o programa para torná-lo mais claro, mas mantenha o mesmo comportamento no caso de fim de arquivo ou estouro da área de armazenamento. Este comportamento é o mais razoável?

1.11 Sumário

Neste ponto cobrimos o que pode ser chamado de núcleo convencional de C. Com essa quantidade de ferramentas, é possível escrever programas úteis de tamanho considerável, e é provavelmente uma boa idéia parar algum tempo para fazê-lo. Os exercícios seguintes sugerem programas um tanto mais complexos que aqueles apresentados neste capítulo.

Depois de você ter adquirido fluência sobre as partes de C aqui expostas, valerá a pena continuar a leitura, pois as facilidades cobertas pelos próximos poucos capítulos revelarão a potência e o poder de expressão da linguagem.

Exercício 1-19. Escreva um programa `destab` que troque caracteres de tabulação da entrada pelo número apropriado de espaços para atingir o próximo ponto de tabulação. Assuma um conjunto fixo de pontos de tabulação, digamos a cada n posições.

Exercício 1-20. Escreva um programa `tab` que troque cadeias de brancos pelo número mínimo de caracteres de tabulação e de brancos, de forma a obter o mesmo espaçamento. Use os mesmos de pontos de tabulação do programa `destab`.

Exercício 1-21. Escreva um programa para “dobrar” linhas longas após o último caractere não branco que ocorra antes da n -ésima coluna de entrada, onde n é um parâmetro. Faça com que seu programa funcione inteligentemente com linhas muito longas e no caso de não haver espaços ou caracteres de tabulação antes da coluna especificada.

Exercício 1-22. Escreva um programa para remover todos os comentários de um programa C. Não se esqueça de manipular cadeias e constantes do tipo caractere entre aspas de forma correta.

Exercício 1-23. Escreva um programa para verificar num programa C erros rudimentares de sintaxe, tais como parênteses, colchetes e chaves não balanceados. Não se esqueça de aspas, apóstrofes e comentários. (Este programa é difícil, se for feito em toda generalidade.)

Capítulo 2

TIPOS, OPERADORES E EXPRESSÕES

Variáveis e constantes são os objetos de dados básicos manipulados em um programa. As declarações listam as variáveis a serem usadas, e definem os tipos que as mesmas devem ter e talvez seus valores iniciais. Os operadores especificam o que deve ser feito com as variáveis. As expressões combinam variáveis e constantes para produzir novos valores. Esses são os tópicos deste capítulo.

2.1 Nomes de Variáveis

Embora não o tenhamos dito de imediato, há algumas restrições aos nomes de variáveis e constantes simbólicas. Os nomes são construídos com letras e dígitos; o primeiro caractere deve ser uma letra. O caractere de sublinha “_” vale como uma letra; ele é útil para aumentar a clareza de nomes muito longos de variáveis. Letras em maiúsculo e minúsculo são diferentes; uma prática tradicional em C é a de usar letras minúsculas para variáveis e maiúsculas para constantes simbólicas.

Somente os primeiros oito caracteres de um nome interno são significativos, embora mais caracteres possam ser usados. Para nomes externos, tais como funções e variáveis externas, o número de caracteres pode ser menor que oito, porque nomes externos são usados por vários montadores e carregadores. O apêndice A fornece detalhes. Além disso, palavras-chave tais como *if*, *else*, *int*, *float*, etc., são *reservadas*: você não pode usá-las como nomes de variáveis. (Elas devem estar em letras minúsculas.)

Naturalmente é melhor escolher nomes de variáveis que tenham algum significado, que estejam relacionados com a finalidade da variável, e que sejam diferenciados tipograficamente.

2.2 Tipos de Dados e Tamanhos

Há poucos tipos de dados em C:

<code>char</code>	um único <i>byte</i> , capaz de conter um caractere do conjunto de caracteres local.
<code>int</code>	um inteiro, tipicamente refletindo o tamanho natural de inteiros da máquina hospedeira.
<code>float</code>	ponto flutuante em precisão simples.
<code>double</code>	ponto flutuante em dupla precisão.

Além do mais, há um número de qualificadores que podem ser aplicados a inteiros: *short*, *long* e *unsigned*. *short* e *long* referem-se a diferentes tamanhos de inteiros. Nume-

ros unsigned obedecem à lei da aritmética módulo 2^n , onde n é o número de bits em um int; números unsigned são sempre positivos. A declaração dos qualificadores tem o formato:

```
short int x;
long int y;
unsigned int z;
```

A palavra int pode ser (e normalmente é) omitida em tais situações.

A precisão desses objetos depende da máquina disponível; a tabela abaixo mostra alguns valores representativos.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

A intenção é que short e long devam prover tamanhos diferentes de inteiros quando praticável; int reflete normalmente o tamanho mais "natural" para uma máquina particular. Como você pode ver, cada compilador é livre para interpretar short e long do modo que for mais apropriado para seu hardware. Uma das únicas coisas que você pode supor é que short não é maior que long.

2.3 Constantes

As constantes int e float já foram expostas, salvo a observação de que a notação científica

123.456e-7

ou

0.12E3

para float são válidas também. Toda constante de ponto flutuante é presumida ser double, de modo que a notação "e" serve para ambos float e double.

Constantes do tipo long são escritas no estilo 123L. Uma constante inteira ordinária que não couber num int é assumida como sendo do tipo long.

Há uma notação para constantes octais e hexadecimais: um O (zero) inicial em uma constante int a define como octal, um 0x ou 0X iniciais a define como hexadecimal. Por exemplo, o número decimal 31 pode ser escrito como 037 em octal e 0x1f ou 0X1F em hexadecimal. Constantes hexadecimais e octais podem também ser seguidas por L para torná-las do tipo long.

Uma *constante do tipo caractere* é um único caractere escrito entre apóstrofes, como 'x'. O valor de uma constante do tipo caractere é o valor numérico do caractere no conjunto de caracteres da máquina. Por exemplo, no conjunto de caracteres ASCH o

caractere zero, ou '0', é 48, e em EBCDIC é 240, ambas são diferentes do valor numérico 0. O uso de '0' ao invés de 48 ou 240 torna o programa independente do valor particular. Constantes do tipo caractere participam em operações numéricas como um número qualquer, embora elas sejam mais frequentemente usadas em comparações com outros caracteres. Uma seção posterior trata das regras de conversão.

Certos caracteres não-gráficos podem ser representados em constantes do tipo caractere por seqüências de escape como \n (nova linha), \t (caractere de tabulação), \0 (caractere nulo), \\ (contrabarra), \' (apóstrofo), etc.; essas seqüências de escape parecem ser dois caracteres, mas representam um só. Além do mais, um padrão arbitrário do tamanho de um byte pode ser gerado escrevendo-se

`'\ddd'`

onde *ddd* tem de um a três dígitos octais, como no seguinte exemplo

```
#define ALIMFORM '\014' /* alimentação de formulário ASCII */
```

A constante do tipo caractere '\0' representa o caractere com valor zero. '\0' é frequentemente usado ao invés de 0 para enfatizar a natureza de caractere de alguma expressão.

Uma *expressão constante* é uma expressão que envolve somente constantes. Tais expressões são avaliadas em tempo de compilação, e não em tempo de execução, e podem ser usadas onde uma constante possa aparecer, tal como em:

```
#define MAXLINHA 1000
char linha[MAXLINHA + 1];
```

ou

```
segundos = 60 * 60 * horas;
```

Uma *constante do tipo cadeia* é uma seqüência de zero ou mais caracteres entre aspas, tal como:

```
"Eu sou uma cadeia"
```

ou

```
"" /* uma cadeia nula */
```

As aspas não fazem parte da cadeia, mas servem somente para delimitá-la. As mesmas seqüências de escape usadas para constantes do tipo caractere podem ser usadas em cadeias; \" representa o caractere aspas.

Tecnicamente, uma cadeia é um arranjo cujos elementos são caracteres. O compilador coloca automaticamente o caractere nulo no fim de cada cadeia de modo que os programas possam encontrar o fim de uma cadeia de forma conveniente. Essa representação implica que uma cadeia não tem limite de tamanho, embora programas tenham de percorrê-la completamente para determiná-lo. O armazenamento físico necessário para uma cadeia é de um byte a mais do que o número de caracteres escritos entre aspas. A seguinte função `strlen(s)` retorna o tamanho de uma cadeia de caracteres *s*, incluindo o \0 terminal.

```

strlen(s) /* retorna tamanho de s */
char s [ ];
{
    int i;

    i = 0;
    while (s[i] != '\0')
        ++ i;
    return (i);
}

```

Seja cuidadoso ao distinguir uma constante do tipo caractere de uma cadeia que contenha um único caractere: 'x' não é igual a "x". A primeira é uma constante do tipo caractere, usada para produzir o valor numérico da letra x no conjunto de caracteres da máquina. A última é uma cadeia de caracteres contendo um caractere (a letra x) e um \0.

2.4 Declarações

Todas as variáveis devem ser declaradas antes de usadas, embora certas declarações possam ser feitas implicitamente pelo contexto. Uma declaração especifica um tipo, e é seguida por uma lista de uma ou mais variáveis daquele tipo, tal como no seguinte exemplo:

```

int inicio, fim, incr;
char c, linha [1000];

```

Variáveis podem ser distribuídas entre declarações livremente; as listas acima poderiam ser igualmente escritas como:

```

int   inicio;
int   fim;
int   incr;
char  c;
char  linha [1000];

```

Essa última forma ocupa mais espaço, mas é mais conveniente para acrescentar um comentário a cada declaração ou para modificações subseqüentes.

Variáveis podem também ser inicializadas em suas declarações, embora haja algumas restrições. Se o nome for seguido por um sinal de igualdade e por uma constante, isto serve para inicializá-lo tal como em:

```

char  contrabarra = '\\';
int    i = 0;
float  eps = 1.0e-5;

```

Se a variável em questão for externa ou estática, a inicialização é feita somente uma vez, conceitualmente antes do programa começar. Variáveis automáticas inicializadas explicitamente serão inicializadas cada vez que a função que as contém for chamada. Variáveis automáticas sem inicialização explícita têm valores iniciais indefinidos (i.e., lixo). Variáveis externas e estáticas são inicializadas com zero, caso não haja inicialização explícita, embora um estilo correto de programação requiera a inicialização.

Discutiremos o assunto de inicialização posteriormente quando novos tipos de dados forem introduzidos.

2.5 Operadores Aritméticos

Os operadores aritméticos binários são: +, -, *, /, e o operador módulo %. Há um operador unário -, mas não existe um + unário.

A divisão inteira trunca qualquer parte fracionária. A expressão

$x \% y$

produz o resto da divisão de x por y, e é igual a zero se y dividir x exatamente. Por exemplo um ano é bissexto se for divisível por 4 mas não por 100, exceto quando for divisível por 400. Assim, temos

```
if (ano % 4 == 0 && ano % 100 != 0 || ano % 400 == 0)
    ano e bissexto
else
    ano nao e bissexto
```

O operador % não pode ser aplicado a float ou double.

Os operadores + e - têm a mesma precedência, a qual é menor que a precedência (idêntica) de *, / e %, que por sua vez, é menor que a do menos unário. Os operadores aritméticos se associam da esquerda para a direita (uma tabela no fim deste capítulo sumariza a precedência e a associatividade de todos os operadores). A ordem de avaliação não é especificada para operadores associativos como * e +; o compilador pode rearrumar uma expressão parentetizada envolvendo um desses operadores. Portanto $a + (b + c)$ pode ser avaliada como $(a + b) + c$. Isso raramente faz alguma diferença mas se uma ordem particular é desejada, variáveis temporárias explícitas podem ser usadas.

A ação tomada no caso de transbordo positivo ou negativo depende da máquina que está sendo utilizada.

2.6 Operadores Relacionais e Lógicos

Os operadores relacionais são

> >= < <=

Eles têm a mesma precedência: Logo abaixo na precedência vêm os operadores

== !=

que têm a mesma precedência. Operadores relacionais têm menor precedência que operadores aritméticos, de forma que expressões tais como $i < \text{lim} - 1$ são avaliadas como $i < (\text{lim} - 1)$, como seria esperado.

Mais interessantes são os conectores lógicos && e ||. Expressões conectadas por && ou || são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Essas propriedades são críticas na escrita de programas que funcionem. Por exemplo, segue um laço da função lelinha que nós escrevemos no Capítulo 1.

```
for (i = 0; i < lim - 1 && {c = getchar ( )} != '\n' && c != EOF; ++ i)
    s[i] = c;
```

É claro que antes de ler um novo caractere é necessário verificar se há espaço para arma-

zená-lo no arranjo `s`; portanto, o teste `i < lim - 1` deve ser feito primeiro. Além disso, se esse teste falhar, não poderemos prosseguir e ler outro caractere.

De forma semelhante, seria desastroso se `c` fosse testado contra EOF antes de `getchar` ser chamada: a chamada deve ocorrer antes do caractere em `c` ser testado.

A precedência de `&&` é maior que a de `|`, e ambas têm menor precedência que os operadores relacionais e de igualdade, de forma que expressões tais como:

```
i < lim - 1 && (c = getchar ( )) != '\n' && c != EOF
```

não necessitam de parênteses extras. Mas desde que a precedência de `!=` é maior que a da atribuição, parênteses são necessários em:

```
(c = getchar ( )) != '\n'
```

para obter o resultado desejado.

O operador unário de negação `!` converte um operando diferente de zero ou verdadeiro no valor 0, e um operando zero ou falso no valor 1. O operador `!` é comumente empregado em construções do tipo:

```
if (! empalavra)
```

ao invés de

```
if (empalavra == 0)
```

É difícil generalizar sobre que forma é melhor. Construções tais como `! empalavra` são lidas mais agradavelmente ("se não em palavra"), mas algumas construções mais complexas podem ser difíceis de entender.

Exercício 2-1. Escreva um laço equivalente ao laço `for` acima sem usar `&&`.

2.7 Conversões de Tipos

Quando operandos de tipos diferentes aparecem em expressões, são convertidos para um tipo comum de acordo com um pequeno número de regras. Em geral, as únicas conversões que ocorrem automaticamente são aquelas que fazem sentido, tal como converter um inteiro para ponto flutuante em uma expressão do tipo `f + i`. Expressões que não fazem sentido, tais como usar um `float` como um subíndice, não são permitidas.

Primeiro, operandos do tipo `char` e `int` podem ser misturados livremente em expressões aritméticas: todo `char` em uma expressão é automaticamente convertido para um `int`. Isso permite uma flexibilidade considerável em certos tipos de transformações de caracteres. Uma dessas transformações é exemplificada pela função `atoi`, que converte uma cadeia de dígitos em seu valor numérico correspondente.

```
atoi (s) /* converte s a um inteiro */
char s [ ];
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++ i)
        n = 10 * n + s[i] - '0';
    return (n);
}
```

Como falamos no Capítulo 1, a expressão

`s[i] - '0'`

dá o valor numérico do caractere armazenado em `s[i]` porque os valores de '0', '1', etc., formam uma sequência contígua, crescente e positiva.

Um outro exemplo de uma conversão de char para int é a função minúsculo que mapeia um caractere para minúsculo *considerando-se apenas o conjunto de caracteres ASCII*. Se o caractere não é maiúsculo, minúsculo o retorna inalterado.

```
minúsculo (c) /* converte c a minúsculo; so para ASCII */
int c;
{
    if (c >= 'A' && c <= 'Z')
        return (c + 'a' - 'A');
    else
        return (c);
}
```

Isso funciona para ASCII porque as letras maiúsculas e minúsculas estão a uma distância numérica fixa e cada alfabeto é contíguo — não há nada exceto letras entre A e Z. Essa última observação *não* é verdadeira para o conjunto de caracteres EBCDIC (IBM 360/370), de modo que esse código falha em tais sistemas — ele converte mais do que letras.

Há um ponto delicado sobre a conversão de caracteres para inteiros. A linguagem não especifica se variáveis do tipo char são valores com ou sem sinal. Quando um char é convertido para um int, pode-se produzir um número *negativo*? Infelizmente, isso varia de máquina para máquina, refletindo diferenças de arquitetura. Em algumas máquinas (PDP-11, por exemplo), um char cujo bit mais significativo for 1 será convertido em um inteiro negativo (extensão do sinal). Em outras máquinas, um char é convertido a int pela adição de zeros à esquerda, e é então sempre positivo.

A definição de C garante que qualquer caractere no conjunto de caracteres-padrão da máquina nunca será negativo, de modo que esses caracteres podem ser usados livremente em expressões como quantidades positivas. Porém padrões arbitrários de bits armazenados em variáveis do tipo caractere podem aparecer como negativos em algumas máquinas, ainda que positivos em outras.

A ocorrência mais comum dessa situação é quando o valor -1 é usado para EOF. Considere o código:

```
char c;

c = getchar ( );
if (c == EOF)
    ...
```

Em uma máquina que não faz extensão de sinal, c é sempre positivo porque ele é um char, ainda que EOF seja negativo. Como resultado, o teste sempre falha. Para evitar isso, nós fomos cuidadosos em usar int ao invés de char para qualquer variável que contenha um valor retornado por getchar.

A razão verdadeira para usar int ao invés de char não está relacionada com qualquer questão de uma possível extensão de sinal. É simplesmente porque getchar deve retornar

todos os possíveis caracteres (de modo que ela possa ser usada para ler qualquer entrada) e, além disso, um valor distinto para EOF. Então seu valor *não pode* ser representado como um char, mas deve ser armazenado como um int.

Uma outra forma útil de conversão automática de tipos é a de expressões relacionais do tipo $i > j$ e expressões lógicas conectadas por $\&\&$ e $||$ que são definidas por ter valor 1 caso verdadeiras e 0 caso falsas. Então a atribuição

```
eh-digito = c >= '0' && c <= '9';
```

atribui a eh-digito o valor 1 se c é um dígito, e 0 caso contrário. (Na parte de teste de if, while, for, etc. "verdadeiro" significa "diferente de zero".)

Conversões aritméticas implícitas funcionam como o esperado. Em geral, se um operador como + ou * que aceite dois operandos (um "operador binário"), tiver operandos de tipos diferentes, o tipo "menor" é *promovido* ao tipo "maior" antes da operação prosseguir. O resultado é do tipo maior. Mais precisamente, para cada operador aritmético, a seguinte sequência de regras de conversão é aplicada.

char e short são convertidos para int, e float é convertido para double.

Então se um ou outro operando for double, o outro é convertido para double e o resultado é double.

Senão, se um ou outro operando for long, o outro é convertido para long e o resultado é long.

Senão, se um ou outro operando for unsigned, outro é convertido para unsigned e o resultado é unsigned.

Senão, os operandos devem ser int, e o resultado é int.

Observe que todo float em uma expressão é convertido para double; toda aritmética de ponto flutuante em C é feita em precisão dupla.

As conversões são feitas em atribuições; o valor do lado direito é convertido para o tipo do lado esquerdo, o qual é o tipo do resultado. Um caractere é convertido para inteiro, por extensão de sinal ou não, como descrito acima. A operação inversa int para char, é bem comportada — os bits excedentes mais significativos são simplesmente descartados. Então em

```
int i;  
char c;
```

```
i = c;  
c = i;
```

o valor de c não é alterado. Isso é verdadeiro com ou sem extensão de sinal.

Se x é float e i é int, então

```
x = i;
```

e

```
i = x;
```

causam conversão; float para int causa truncamento da parte fracionária. double é convertido para float por arredondamento. Inteiros longos são convertidos para inteiros mais curtos ou para char pela remoção dos bits mais significativos.

Visto que o argumento de uma função é uma expressão, a conversão de tipos ocorrerá quando os argumentos forem passados para a função: em particular, `char` e `short` tornam-se `int`, e `float` torna-se `double`. É por isso que declaramos argumentos de funções como `int` e `double` mesmo quando a função é chamada com `char` e `float`.

Finalmente, a conversão explícita de tipos pode ser forçada, em qualquer expressão, com uma construção chamada *molde*. Na construção

(nome-de-tipo) expressão

a *expressão* é convertida para o tipo especificado pelas regras de conversão acima. O significado preciso de um molde é de fato como se a *expressão* fosse atribuída a uma variável do tipo especificado, que é então usada no lugar da construção. Por exemplo, a rotina de biblioteca `sqrt` espera um `double` como argumento, e dará um resultado sem sentido se outro argumento lhe for passado. Assim, se `n` é um inteiro,

`sqrt ((double) n)`

converte `n` para `double` antes de passá-lo para `sqrt` (observe que o molde produz o *valor* de `n` no tipo apropriado; o conteúdo de `n` não é alterado). O operador de molde tem a mesma precedência que qualquer outro operador unário, como está resumido na tabela no fim deste capítulo.

Exercício 2-2. Escreva a função `htoi(s)`, a qual converte uma cadeia de dígitos hexadecimais no seu valor inteiro equivalente. Os dígitos permitidos são de 0 a 9, de `a` a `f`, e de `A` a `F`.

2.8 Operadores de Incremento e Decremento

C fornece dois operadores não usuais para incrementar e decrementar variáveis. O operador de incremento `++` soma 1 ao seu operando; o operador de decremento `--` subtrai 1. Usamos frequentemente o operador `++`, como em:

```
if (c == '\n')
    ++nl;
```

O aspecto não usual é que `++` e `--` podem ser usados tanto como operadores prefixados (antes da variável, como em `++n`) ou pós-fixados (após a variável, como em `n++`). Em ambos os casos, a variável `n` é incrementada. Porém a expressão `++n` incrementa `n` *antes* de usar seu valor, enquanto `n++` incrementa `n` *após* seu valor ser usado. Isso significa que num contexto onde o valor é usado (além de obter o efeito), `++n` e `n++` são diferentes. Se `n` é 5, então

```
x = n ++;
```

atribui 5 a `x`, mas

```
x = ++ n;
```

atribui 6 a `fluxff`. Em ambos os casos, `n` torna-se 6. Os operadores de incremento e decremento podem ser aplicados somente a variáveis; uma expressão do tipo `x = (i + j) ++` é ilegal.

Num contexto onde o valor não é usado e queremos apenas o efeito de incrementar, tal como em:

```

if (c == '\\,')
    nl ++;

```

a escolha de prefixação ou posfixação é de acordo com o gosto. Mas há situações onde um ou outro uso é mais apropriado. Por exemplo, considere a função `comprime (s, c)` que remove toda ocorrência do caractere `c` da cadeia `s`.

```

comprime (s, c) /* deleta todos os c de s */
char s[ ];
int c;
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}

```

Cada vez que um caractere diferente de `c` ocorre em `s`, ele é copiado para a `j`-ésima posição, e somente então `j` é incrementado para ficar pronto para o próximo caractere. Isso é exatamente equivalente a

```

if (s[i] != c) {
    s[j] = s[i];
    j++;
}

```

Um outro exemplo de uma construção semelhante vem da função `lelinha` que escrevemos no Capítulo 1, onde podemos trocar

```

if (c == '\n') {
    s[i] = c;
    ++i;
}

```

por algo mais compacto

```

if (c == '\n')
    s[i++] = c;

```

Como terceiro exemplo, a função `strcat (s, t)` concatena a cadeia `t` ao fim da cadeia `s`. `strcat` assume que há espaço suficiente em `s` para conter a combinação.

```

strcat (s, t) /* concatena t ao fim de s */
char s[ ], t[ ]; /* deve caber em s */
{
    int i, j;

```

```

i = j = 0;
while (s[j] != '\0') /* acha o fim de s */
    i++;
while ((s[i++] = t[j++]) != '\0') /* copia t */
    ;
}

```

A cada caractere copiado de *t* para *s*, o `++` posfixado é aplicado a *i* e *j* para garantir que eles estejam em posição para o próximo passo do laço.

Exercício 2-3. Escreva uma versão alternativa de `comprime (s1, s2)` que remova cada caractere de *s1* que se case a algum caractere presente na cadeia *s2*.

Exercício 2-4. Escreva uma função qualquer (*s1, s2*) que retorne a primeira posição na cadeia *s1* onde qualquer caractere presente na cadeia *s2* ocorra, ou `-1` se *s1* não contiver caracteres de *s2*.

2.9 Operadores Lógicos Bit-a-Bit

C provê um número de operadores para manipulação de bit; eles não podem ser aplicados a `float` e `double`.

```

&    E bit-a-bit
|    OU inclusivo bit-a-bit
^    OU exclusivo bit-a-bit
<<  deslocamento a esquerda
>>  deslocamento a direita
~    complemento de um (unário)

```

O operador `&` é frequentemente usado para mascarar algum conjunto de bits; por exemplo,

```
c = n & 0177;
```

zera todos os bits de *n*, exceto os 7 menos significativos. O operador bit-a-bit OU `|` é usado para ligar bits:

```
x = x | MASC;
```

muda para um em *x* os bits que estão ligados (i.e., são iguais a um) em `MASC`.

Você deve distinguir cuidadosamente os operadores bit-a-bit `&` e `|` dos conectores lógicos `&&` e `||`, que implicam numa avaliação da esquerda para a direita de um valor booleano. Por exemplo, se *x* é 1 e *y* é 2, então *x* & *y* é zero, enquanto que *x* && *y* é um. (Por quê?)

Os operadores de deslocamento `<<` e `>>` deslocam à esquerda e à direita o seu operando esquerdo pelo número de bits especificados pelo operando direito. Então *x* `<<` 2 desloca *x* à esquerda de 2 bits, preenchendo os bits vagos com 0; isso é equivalente a multiplicar *x* por 4. Um deslocamento à direita de uma quantidade `unsigned` preenche os bits vagos com 0. Um deslocamento à direita de uma quantidade com sinal preenche os bits vagos com o bit de sinal (deslocamento aritmético) em algumas máquinas como o PDP-11, e com 0 (deslocamento lógico) em outras.

O operador unário $\sim$ fornece o “complemento de um” de um inteiro, isto é, ele converte cada bit 1 em 0 e vice-versa. Esse operador é tipicamente usado em expressões do tipo

$x \& \sim 077$

que zera os últimos seis bits de x . Observe que $x \& \sim 077$ é independente de tamanho de palavra, e é preferível a, por exemplo, $x \& 0177700$, que supõe que x tenha 16 bits. A forma transportável não envolve custo extra, desde que ~ 077 é uma expressão constante e é avaliada em tempo de compilação.

Para ilustrar o uso de alguns operadores bit-a-bit, considere a função `obtembits` (x , p , n) que retorna (ajustado à direita) o campo de n bits que começam na p -ésima posição. Nós assumimos que o bit 0 é o mais à direita e que n e p são sempre valores positivos sensatos. Por exemplo, `obtembits` (x , 4, 3) retorna o valor dos três bits nas posições 4, 3 e 2, ajustados à direita.

```
obtembits (x, p, n) /* obtem n bits da posicao p */
unsigned x, p, n;
{
    return ((x >> (p + 1 - n)) & ~ (~0 << n));
}
```

$x >> (p + 1 - n)$ move o campo desejado para a extremidade direita da palavra. A declaração de x como `unsigned` garante que, quando ele é deslocado para a direita, os bits vagos sejam preenchidos com zeros, e não com bits de sinal, independente da máquina onde o programa for executado. $\sim 0 << n$ é equivalente a todos os bits iguais a 1; deslocando-os à esquerda n bits com $\sim 0 << n$, produzimos uma máscara de zeros nos n bits mais à direita e com os bits restantes iguais a um; a complementação com $\sim$ cria uma máscara com bits iguais a um nas n posições mais à direita.

Exercício 2-5. Modifique `obtembits` para numerar os bits da esquerda para a direita.

Exercício 2-6. Escreva uma função `comp_pal` () que calcule o tamanho da palavra da máquina hospedeira, isto é, o número de bits num int. A função deve ser transportável, de forma que o mesmo código-fonte funcione em todas as máquinas.

Exercício 2-7. Escreva uma função `gira-dir` (n , b) que gire o inteiro n à direita de b bits.

Exercício 2-8. Escreva uma função `inverte` (x , p , n) que inverta (isto é, mude 1 em 0 e vice-versa) n bits de x começando no p -ésimo bit, deixando os demais inalterados.

2.10 Operadores e Expressões de Atribuição

Expressões do tipo

$i = i + 2$

em que o lado esquerdo é repetido à direita podem ser escritos de forma mais compacta

$i += 2$

usando um *operador de atribuição* tal como $+=$. A maioria dos operadores binários (tais como $+$ que tem um operando à esquerda e outro à direita) tem um operador de atribuição correspondente $op=$, onde op pode ser um dos seguintes:

+ - * / % << >> & ~ |

Se *e1* e *e2* são expressões, então

e1 op = e2

é equivalente a

e1 = (e1) op (e2)

exceto que *e1* só é avaliada uma vez. Observe os parênteses em *e2*:

*x * = y + 1*

é equivalente a

*x = x * (y + 1)*

e não a

*x = x * y + 1*

Como exemplo, vejamos a função *contabits* que conta o número de bits 1 no seu argumento inteiro.

```
contabits (n) /* conta os bits ligados em n */
unsigned n;
{
    int b;

    for (b = 0; n != 0; n >>= 1)
        if (n & 01)
            b++;
    return (b);
}
```

Concisão à parte, os operadores de atribuição têm a vantagem de corresponder melhor ao modo de pensar das pessoas. Nós dizemos “soma 2 a *i*” ou “incremente *i* de 2”, e não “toma *i*, soma 2, e coloca o resultado em *i*”. Daí *i* += 2. Além disso, para uma expressão complicada do tipo

yyval[yyvsp[p3+p4] + yypv[p1+p2]] += 2

o operador de atribuição torna o código mais fácil de entender, já que o leitor não tem de analisar a escrita correta de duas expressões longas para verificar se são iguais ou porque não o são. E um operador de atribuição pode até ajudar o compilador a produzir um código mais eficiente.

Já usamos o fato de que o comando de atribuição tem um valor e pode ocorrer em expressões; o exemplo mais comum é

```
while ((c = getchar ( )) != EOF)
```

...

Atribuições usando os outros operadores de atribuição também podem ocorrer em expressões, embora tenham uma frequência menor.

O tipo de uma expressão de atribuição é o tipo do seu operando esquerdo.

Exercício 2-9. Num sistema numérico usando complemento de 2, $x \& (x - 1)$ apaga o bit 1 que está mais à direita em x . (Por quê?) Use essa observação para escrever uma versão mais rápida de `contabits`.

2.11 Expressões Condicionais

Os comandos

```
if (a > b)
    z = a;
else
    z = b;
```

atribuem a z o maior valor entre a e b . A *expressão condicional*, escrita com o operador ternário “?:”, fornece uma forma alternativa de escrever isso e outras construções semelhantes. Na expressão

```
e1 ? e2 : e3
```

a expressão $e1$ é avaliada primeiro. Se for diferente de zero (verdadeira), então a expressão $e2$ é avaliada, e é o valor da expressão condicional. Senão $e3$ é avaliada, e é o valor da expressão condicional. Somente $e2$ ou $e3$ é avaliada. Então para atribuir a z o máximo entre a e b ,

```
z = (a > b) ? a : b; /* z = max (a, b) */
```

Deve-se notar que uma expressão condicional é também uma expressão, e pode ser usada como qualquer outra expressão. Se $e2$ e $e3$ são de tipos diferentes, o tipo do resultado é determinado pelas regras de conversão discutidas inicialmente neste capítulo. Por exemplo, se f for um `float`, e n for um `int`, então

```
(n > 0) ? f : n
```

é do tipo `double`, que n seja positivo ou não.

Parênteses não são necessários ao redor da primeira expressão da expressão condicional, desde que a precedência de “?” é muito baixa, imediatamente acima da atribuição. Os parênteses são aconselháveis de qualquer forma, entretanto, uma vez que eles tornam a parte condicional da expressão mais visível.

A expressão condicional leva freqüentemente a um código compacto. Por exemplo, o laço abaixo imprime N elementos de um arranjo, 10 por linha, com cada coluna separada por um espaço, e com cada linha (inclusive a última) terminada por exatamente um caractere de nova-linha.

```
for (i = 0; i < N; i++)
    printf ("%6d%c", a[i], (i%10 == 9 || i == N - 1) ? '\n' : ' ');
```

Um caractere de nova-linha é impresso após cada conjunto de dez elementos e após o N -ésimo elemento. Todos os outros elementos são seguidos por um espaço. Embora isso possa parecer complicado, é instrutivo tentar escrevê-lo sem expressão condicional.

Exercício 2-10. Reescreva a função `minusc` a qual converte letras maiúsculas em minúsculas, com uma expressão condicional no lugar de `if-else`.

2.12 Precedência e Ordem de Avaliação

A tabela abaixo sumariza as regras de precedência e de associatividade de todos os operadores, inclusive aqueles que nós não discutimos ainda. Os operadores numa mesma linha têm a mesma precedência; as linhas estão em ordem decrescente de precedência de forma que, por exemplo, *, /, e % têm a mesma precedência, a qual é maior que a de + e -.

Operador	Associatividade
() [] -> .	esquerda para direita
! ~ ++ -- (type) * & sizeof	direita para esquerda
* / %	esquerda para direita
+ -	esquerda para direita
<< >>	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
& ^	esquerda para direita
	esquerda para direita
&&	esquerda para direita
	esquerda para direita
?:	direita para esquerda
= += -= etc.	direita para esquerda
, (Capítulo 3)	esquerda para direita

Os operadores - > e . são usados para acessar membros de estruturas; eles serão vistos no Capítulo 6, juntamente com sizeof (tamanho de um objeto). O Capítulo 5 discute * (indireção) e & (endereço). Observe que a precedência dos operadores lógicos bit-a-bit &, |, e ^ vêm abaixo da de == e !=. Isso implica que expressões de testes de bits do tipo

```
if ((x & MASC) == 0) ...
```

devem ser parentetizadas para darem resultados apropriados.

Como mencionado antes, expressões envolvendo um dos operadores associativos e comutativos (*, +, &, ^, |) podem ser rearrumados mesmo quando parentetizados. Em muitos casos isso não faz diferença; em situações onde pode haver diferença, variáveis temporárias explícitas podem ser usadas para forçar uma ordem particular de avaliação.

C, como a maioria das linguagens, não especifica em que ordem os operandos de um operador são avaliados. Por exemplo, num comando do tipo:

```
x = f() + g();
```

f pode ser avaliada antes de g ou vice-versa; então, se tanto f ou g alterar uma variável externa da qual o outro dependa, x pode depender da ordem de avaliação. Novamente, re-

sultados intermediários podem ser armazenados em variáveis temporárias para assegurar uma sequência particular.

De forma semelhante, a ordem em que os argumentos de funções são avaliados não é especificada, de modo que o comando

```
printf ("%d %d\n", ++n, pot (2, n)); /* ERRADO */
```

pode (e vai) produzir resultados diferentes em máquinas diferentes, dependendo se *n* é ou não incrementado antes da chamada a *pot*. A solução é escrever:

```
++n;  
printf ("%d %d\n", n, pot (2, n));
```

Chamadas de funções, atribuições aninhadas, e operadores de incremento e de decremento causam “efeitos colaterais” — alguma variável é mudada como resultado da avaliação de uma expressão. Em qualquer expressão envolvendo efeitos colaterais, pode haver dependências sutis na ordem em que as variáveis que tomam parte na expressão são armazenadas. Uma situação infeliz é exemplificada pelo comando:

```
a[i] = i ++;
```

A questão é se o subíndice usado é o velho ou o novo valor de *i*. O compilador pode fazer isso de maneiras diferentes, e gerar respostas diferentes dependendo da sua interpretação.

O compilador decide quando os efeitos colaterais (a atribuição de valores a variáveis) serão aplicados, porque a melhor ordem depende da arquitetura da máquina.

A moral dessa discussão é que a escrita de código com dependência na ordem de avaliação de expressões não é boa prática de programação em qualquer linguagem. Naturalmente, é necessário conhecer o que evitar, mas se você não conhece *como* certas coisas são feitas em várias máquinas, esta inocência pode ajudar a protegê-lo. (O verificador *C lint* detecta a maioria das dependências na ordem de avaliação.)

Capítulo 3

FLUXO DE CONTROLE

Os comandos de fluxo de controle de uma linguagem especificam a ordem em que a computação é feita. Nós já vimos as construções mais comuns de fluxo de controle de C nos exemplos anteriores; aqui completaremos o conjunto e seremos mais precisos na discussão das construções já vistas.

3.1 Comandos e Blocos

Uma *expressão* tal como $x = 0$ ou $i++$ ou `printf ( . . . )` torna-se um *comando* quando seguida por um ponto-e-vírgula, como em:

```
x = 0;
i++;
printf ( . . . );
```

Em C, o ponto-e-vírgula é o terminador de comandos, e não um separador como em linguagens do tipo ALGOL.

As chaves { e } são usadas para agrupar declarações e comandos num *comando composto* ou *bloco* de modo que são sintaticamente equivalentes a um único comando. As chaves que parentizam os comandos de uma função são um exemplo óbvio; chaves parentizando múltiplos comandos após um `if`, `else`, `while`, ou `for` são outro. (Variáveis podem ser declaradas em *qualquer* bloco; falaremos sobre isso no Capítulo 4). Nunca há um ponto-e-vírgula após a chave direita que termina um bloco.

3.2 If-Else

O comando `if-else` é usado para tomar decisões. Formalmente, a sintaxe é:

```
if (expressão)
    comando-1
else
    comando-2
```

onde a parte `else` é opcional. A *expressão* é avaliada; se for “verdadeira” (isto é, se *expressão* tiver um valor diferente de zero), o *comando-1* é executado. Se for “falsa” (*expressão* tiver um valor igual a zero) e se houver uma parte `else`, o *comando-2* é executado.

Desde que um `if` simplesmente testa o valor numérico de uma expressão, certas abreviações na codificação são possíveis. A mais óbvia é escrever

```
if (expressao)
```

ao invés de

```
if (expressao != 0)
```

Algumas vezes isso é natural e claro; outras vezes, é totalmente obscuro.

Devido à parte `else` do `if-else` ser opcional, há uma ambigüidade quando um `else` é omitido em uma sequência de comandos `if` aninhados. Isso é resolvido do modo usual – o `else` é sempre associado com o mais recente `if` sem `else`. Por exemplo, em

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

o `else` corresponde ao `if` interno, como mostrado pela endentação. Caso isto não seja o que você quer, chaves devem ser usadas para forçar a associação apropriada:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

A ambigüidade é especialmente perniciosa em situações tais como:

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf ("... ");
            return (i);
        }
    else /* errado */
        printf ("erro - n e igual a zero\n");
```

A endentação mostra inequivocamente o que você quer, mas o compilador não entende e associa o `else` ao `if` interno. Esse tipo de problema pode ser muito difícil de achar.

Observe que há um ponto-e-vírgula após `z = a` em

```
if (a > b)
    z = a;
else
    z = b;
```

Isso é porque, gramaticamente, um *comando* segue o `if`, e um comando do tipo expressão como `z = a` sempre termina por um ponto-e-vírgula.

3.3 Else-If

A construção

```
if (expressao)
    comando
```

```

else if (expressao)
    comando
else if (expressao)
    comando
else
    comando

```

ocorre tão freqüentemente que merece uma discussão em separado. Essa sequência de if's é a forma mais geral de escrever uma decisão múltipla. As *expressões* são avaliadas em ordem; se qualquer expressão for verdadeira, o *comando* associado a ela é executado, e toda a cadeia é terminada. O código para cada *comando* é um comando único ou um grupo entre chaves.

A última parte *else* trata do caso "nenhuma das acima" ou caso default quando nenhuma das condições é satisfeita. Algumas vezes não há ação explícita para o default; nesse caso o

```

else
    comando

```

final pode ser omitido, ou pode ser usado para a verificação de erro, capturando assim uma condição impossível.

Para ilustrar uma decisão tripla, segue uma função de pesquisa binária que decide se um valor particular de *x* ocorre num arranjo ordenado *v*. Os elementos de *v* devem estar em ordem crescente. A função retorna a posição (um número entre 0 e *n* - 1) se *x* ocorre em *v*, e -1 caso contrário.

```

pesq_binaria (x, v, n) /* acha x em v[0] ... v[n-1] */
int x, v [ ], n;
{
    int inicio, fim, meio;

    inicio = 0;
    fim = n - 1;
    while (inicio <= fim) {
        meio = (inicio + fim) / 2;
        if (x < v [meio])
            fim = meio - 1;
        else if (x > v [meio])
            inicio = meio + 1;
        else /* achou */
            return (meio);
    }
    return (- 1);
}

```

A decisão fundamental é descobrir se *x* é menor, maior, ou igual ao elemento no meio *v*[*meio*] em cada passo; isso é uma construção apropriada para um *else if*.

3.4 Switch

O comando *switch* é uma construção especial de decisão múltipla que testa se uma expressão casa um de vários valores *constantes*, e desvia de acordo com o resultado. No

Capítulo 1 nós escrevemos um programa para contar o número de cada dígito, espaço em branco, e outros caracteres usando uma sequência `if . . . else if . . . else`. Aqui está o mesmo programa com um `switch`.

```
main ( ) /* contar dígitos, espaço branco, outros */
{
    int c, i, nbranco, noutro, ndigito [10];

    nbranco = noutro = 0;
    for (i = 0; i < 10; i++)
        ndigito [i] = 0;

    while ((c = getchar ( )) != EOF)
        switch (c) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                ndigito [c - '0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nbranco++;
                break;
            default:
                noutro++;
                break;
        }

    printf ("dígitos =");
    for (i = 0; i < 10; i++)
        printf (" %d", ndigito [i]);
    printf ("\nespaço branco = %d, outros = %d\n",
        nbranco, noutro);
}
```

O `switch` avalia a expressão inteira entre parênteses (neste programa, o caractere `c`) e compara seu valor com todos os casos. Cada caso deve ser rotulado por uma constante do tipo inteiro ou caractere ou por uma expressão constante. Se um caso for igual ao valor da expressão, a execução começa nele. O caso rotulado `default` é executado se nenhum dos outros casos for satisfeito. Um `default` é opcional; se não houver um e nenhum dos casos

for satisfeito, nenhuma ação é tomada. Casos e default podem ocorrer em qualquer ordem. Os casos devem ser todos diferentes.

O comando `break` causa uma saída imediata do `switch`. Devido os casos servirem apenas como rótulos, após a execução do código de um caso, o fluxo *prossigue* para o próximo caso, a menos que você tome uma ação alternativa de escape. `break` e `return` são as formas mais comuns de sair de um `switch`. Um comando `break` pode também ser usado para forçar uma saída imediata de um laço `while`, `for` e `do`, como veremos posteriormente neste capítulo.

Prossiguir de um caso para outro é uma faca de dois gumes. Do lado positivo, ela permite múltiplos casos para uma única ação, como nos casos branco, caractere de tabulação ou nova-linha neste exemplo. Mas implica também que, normalmente, cada caso deva ser encerrado com um `break` para evitar o prosseguimento do fluxo para o próximo caso. Prossiguir de um caso para outro não é robusto, propiciando a desintegração do programa quando o mesmo é modificado. Com a exceção de múltiplos rótulos para uma única computação, o prosseguimento de um caso para outro deve ser usado raramente.

Para obter um bom estilo de programação, coloque um `break` após o último caso (o default neste programa) mesmo que não seja logicamente necessário. Algum dia quando outros casos forem acrescentados, essa pequena precaução ajudá-lo-á.

Exercício 3-1. Escreva uma função `expande (s, t)` que converta caracteres como nova-linha e tabulação em seqüências visíveis como `\n` e `\t`, respectivamente, quando copiar a cadeia `t` em `s`. Use um `switch`.

3.5 Laços — While e For

Nós já vimos os laços `while` e `for`. Em

```
while (expressao)  
    comando
```

a *expressao* é avaliada. Se for diferente de zero, *comando* é executado e *expressao* é reavaliada. Este ciclo continua até *expressao* se tornar zero, continuando então a execução após *comando*.

O comando `for`

```
for (expressao-1; expressao-2; expressao-3)  
    comando
```

é equivalente a

```
expressao-1;  
while (expressao-2) {  
    comando;  
    expressao-3;  
}
```

Gramaticalmente, os três componentes de um `for` são *expressões*. Normalmente, *expressao-1* e *expressao-3* são atribuições ou chamadas de função e *expressao-2* é uma expressão relacional. Qualquer uma das três partes pode ser omitida, embora os ponto-e-vírgula devam permanecer. Se *expressao-1* ou *expressao-3* forem omitidas, ela é simplesmente

desconsiderada. Se o teste, *expressao-2*, não está presente, é considerada permanentemente verdadeira, de forma que

```
for ( ; ) {  
    ...  
}
```

é um laço “infinito”, e presumivelmente será encerrado por outros meios (tais como um *break* ou *return*).

Quando usar *while* ou *for* é uma questão de gosto. Por exemplo, em

```
while ((c = getchar ( )) == ' ' || c == '\n' || c == '\t')  
    ; /* pula espaço em branco */
```

não há inicialização nem reinicialização, de modo que o *while* parece mais natural.

O *for* é claramente superior quando há uma inicialização e reinicialização simples, já que ele guarda os comandos de controle do laço juntos e visíveis no topo do laço. Isso é muito óbvio em

```
for (i = 0; i < N; i + +)
```

que é o idioma C para processar os primeiros *N* elementos de um arranjo, análogo ao laço *DO* do Fortran ou *PL/I*. A analogia não é perfeita, entretanto, desde que os limites de um laço *for* podem ser alterados dentro do laço, e a variável de controle *i* retém seu valor quando o laço termina por alguma razão. Devido aos componentes do *for* serem expressões arbitrárias, laços *for* não são restritos a progressões aritméticas. No entanto, é estilisticamente ruim forçar cálculos não relacionados num *for*; ele é melhor empregado para operações de controle de laços.

Como um exemplo maior, segue uma outra versão da função *atoi* para converter uma cadeia de dígitos no seu valor numérico correspondente. Esta versão é mais geral; ela funciona com espaços iniciais opcionais, e um sinal *+* ou *-* opcional. (O Capítulo 4 mostra a função *atof* que faz a mesma conversão para números de ponto flutuante.)

A estrutura básica do programa reflete o formato de entrada:

```
salte espaços em branco, se houver  
obtenha o sinal, se houver  
obtenha a parte inteira, converta-a
```

Cada passo faz sua parte, e deixa as coisas arrumadas para o próximo passo. O processo inteiro termina no primeiro caractere que não puder fazer parte de um número.

```
atoi (s) /* converte s a um inteiro */  
char s [ ];  
{  
    int i, n, sinal;  
  
    for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i + +)  
        ; /* pula espaço branco */  
    sinal = 1;  
    if (s[i] == '+' || s[i] == '-') /* sinal */  
        sinal = (s[i + +] == '+') ? 1 : - 1;  
    for (n = 0; s[i] >= '0' && s[i] <= '9'; i + +)  
        n = 10 * n + s[i] - '0';
```

```

        return (sinal * n);
    }

```

As vantagens de manter o controle do laço centralizado são mais óbvias quando há vários laços aninhados. A função seguinte é uma ordenação Shell para um arranjo de inteiros. A idéia básica da ordenação Shell é que em estágios iniciais, elementos distantes são comparados, ao invés de elementos adjacentes, como em ordenações simples de trocas. Isto tende a eliminar grande quantidade de desordem rapidamente, de modo que estágios posteriores tenham menos trabalho a fazer. O intervalo entre elementos comparados é gradualmente decrementado até um, em cujo ponto a ordenação se torne efetivamente um método de troca adjacente.

```

shell (v, n) /* ordena v [0] . . . v [n - 1] em ordem crescente */
int v [ ], n;
{
    int inter, i, j, temp;

    for (inter = n/2; inter > 0; inter /= 2)
        for (i = inter; i < n; i++)
            for (j = i - inter; j >= 0 && v [j] > v [j + inter]; j -= inter)
                temp = v [j];
                v [j] = v [j + inter];
                v [j + inter] = temp;
}

```

Há três laços aninhados. O mais externo controla o intervalo entre os elementos comparados partindo de $n/2$ e dividindo por 2 a cada passo até se tornar zero. O laço do meio compara cada par de elementos separados por *inter*; o laço mais interno inverte os elementos que estão fora de ordem. Visto que *inter* é eventualmente reduzido a um, todos os elementos são eventualmente ordenados corretamente. Observe que a generalidade do *for* torna o formato do laço externo igual ao dos outros laços, muito embora ele não seja uma progressão aritmética.

Um último operador em C é a vírgula “,” freqüentemente usada no comando *for*. Um par de expressões separadas por uma vírgula é avaliado da esquerda para a direita, e o tipo e valor do resultado são o tipo e o valor do operando direito. Então num comando *for*, é possível colocar múltiplas expressões em várias partes, por exemplo para processar dois índices em paralelo. Isto é ilustrado na função *inverte (s)*, que inverte a cadeia *s*.

```

inverte (s) /* inverte a cadeia s */
char s [ ];
{
    int c, i, j;

    for (i = 0, j = strlen (s) - 1; i < j; i++, j--) {
        c = s [i];
        s [i] = s [j];
        s [j] = c;
    }
}

```

As vírgulas que separam argumentos de funções, variáveis em declarações, etc., *não* são operadores e *não* garantem a avaliação da esquerda para a direita.

Exercício 3-2. Escreva uma função *expande (s1, s2)* que expanda notações abreviadas tais como *a-z* na cadeia *s1* numa sequência equivalente completa *abc...yz* em *s2*. Permita letras minúsculas e maiúsculas e dígitos, e esteja preparado para manipular casos tais como *a-b-c* e *a-z0-9* e *-a-z* (uma convenção útil é aquela que toma um *-* inicial ou final literalmente).

3.6 Laços – Do-while

Os laços *while* e *for* compartilham o atributo útil de testar a condição de término no início, ao invés de no fim do laço, como discutido no Capítulo 1. O terceiro laço em C, o *do-while*, testa a condição de término no fim, *após* cada passo no corpo do laço; o corpo é sempre executado ao menos uma vez. A sintaxe é:

```
do
    comando
while (expressao);
```

O *comando* é executado, então *expressao* é avaliada. Se for verdadeira, *comando* é executado novamente, e assim por diante. Se a expressão se tornar falsa, o laço termina.

Como poderia ser esperado, o *do-while* é muito menos usado que o *while* e o *for*, somando talvez 5% de todos os laços. No entanto, ele é valioso de vez em quando, como na seguinte função *itoa*, que converte um número para uma cadeia de caractere (o inverso de *atoi*). O trabalho é um pouco mais complicado do que se poderia supor à primeira vista, porque os métodos simples de geração de dígitos os geram na ordem errada. Nós escolhemos gerar a cadeia de trás para a frente e então invertê-la.

```
itoa (n, s) /* converte n para caracteres em s */
char s [ ];
int n;
{
    int i, sinal;

    if ((sinal = n) < 0) /* guarda o sinal */
        n = - n; /* torna n positivo */
    i = 0;
    do { /* gera os dígitos em ordem inversa */
        s[i++] = n % 10 + '0'; /* obtém próximo dígito */
    } while ((n /= 10) > 0); /* remove o dígito */
    if (sinal < 0)
        s[i++] = '-';
    s[i] = '\0';
    inverte (s);
}
```

O *do-while* é necessário, ou pelo menos conveniente, desde que ao menos um caractere seja colocado no arranjo *s*, independentemente do valor de *n*. Nós também usamos chaves para o único comando que compõe o corpo do *do-while* mesmo sendo desnecessário,

de modo que o leitor pouco atento não confunda a parte `while` com o início de um laço `while`.

Exercício 3-3. Numa representação numérica em complemento de 2, nossa versão de `itoa` não manipula o maior número negativo, isto é, o valor de n igual a $-(2 \text{ elevado a tamanho da palavra} - 1)$. Explique por que. Modifique-a para imprimir esse valor corretamente, independentemente da máquina onde for executada.

Exercício 3-4. Escreva a função análoga `itob (n, s)` que converte o inteiro sem sinal n numa cadeia de dígitos binários s . Escreva `itoh`, que converte para hexadecimal.

Exercício 3-5. Escreva uma versão de `itoa` que aceite três argumentos ao invés de dois. O terceiro argumento é o tamanho mínimo do campo; o número convertido deve ser preenchido com espaços à esquerda, se for necessário, de forma a torná-lo do comprimento desejado.

3.7 Break

É conveniente, às vezes, controlarmos a saída de um laço de outro modo além do teste, no início ou no fim do mesmo. O comando `break` permite uma saída antecipada de um `for`, `while`, e `do`, como no `switch`. Um comando `break` faz com que o laço (ou `switch`) mais interno seja terminado imediatamente.

O programa seguinte remove brancos e caracteres de tabulação finais de cada linha da entrada, usando um `break` para sair do laço quando o caractere diferente de branco ou tabulação mais à direita for encontrado.

```
#define MAXLINHA 1000

main ( ) /* remove brancos e tabulacoes finais */
{
    int n;
    char linha[MAXLINHA];

    while ((n = lelinha (linha, MAXLINHA)) > 0) {
        while (-- n >= 0)
            if (linha[n] != ' ' && linha[n] != '\t'
                && linha[n] != '\n')
                break;
        linha[n + 1] = '\0';
        printf ("%s\n", linha);
    }
}
```

`lelinha` retorna o tamanho da linha. O laço `while` mais interno começa no último caractere da linha (lembre-se que `-- n` decrementa n antes de usar seu valor), e pesquisa de trás para a frente procurando o primeiro caractere que não seja branco, uma tabulação ou nova linha. O laço é encerrado quando um tal caractere é encontrado ou quando n torna-se negativo (isto é, quando a linha inteira já foi pesquisada). Você deveria verificar que esse é o comportamento correto mesmo se a linha contiver somente espaços em branco.

Uma alternativa do `break` é pôr o teste no próprio laço:

```
while ((n = telinha (linha, MAXLINHA)) > 0) {
    while (-- n >= 0 &&
        (linha[n] == ' ' ||
         linha[n] == '\t' ||
         linha[n] == '\n'))
        ...
}
```

Esta versão é inferior à versão anterior, porque o teste é mais difícil de entender. Testes que necessitam de uma mistura de `&&`, `|`, `!`, ou parênteses devem ser evitados de modo geral.

3.8 Continue

O comando `continue` é relacionado com o `break`, mas é menos usado; ele inicia a próxima iteração do laço mais interno. No `while` e no `do`, isso significa que a parte do teste será executada imediatamente; no `for`, o controle passa para o passo de reinicialização. (continue se aplica somente a laços, e não a `switch`. Um `continue` num `switch` dentro de um laço provoca a próxima iteração do laço.).

Como um exemplo, o fragmento abaixo processa somente números positivos no arranjo `a`; valores negativos são saltados.

```
for (i = 0; i < N; i++) {
    if (a[i] < 0) /* elementos negativos saltados */
        continue;
    ... /* elementos positivos processados */
}
```

O comando `continue` é frequentemente usado quando a parte do laço que se segue é complicada, de modo que a inversão do teste e a endentação de um nível adicional aninharia o programa muito profundamente.

Exercício 3-6. Escreva um programa que copie sua entrada para sua saída, mas que imprima apenas uma de um grupo de linhas adjacentes idênticas. (Isto é uma versão simples do utilitário *uniq* do UNIX.)

3.9 Goto e Rótulos

C fornece o comando infinitamente abusável `goto`, e rótulos para desvios. Formalmente, o `goto` nunca é necessário, e na prática é sempre fácil escrever código sem usá-lo. Nós não usamos o `goto` neste livro.

No entanto, sugeriremos algumas poucas situações onde o `goto` possa ser útil. O uso mais comum é para abandonar o processamento numa estrutura aninhada profundamente, tal como o caso em que se deseja sair de dois laços ao mesmo tempo. O comando `break` não pode ser usado diretamente já que abandona somente o laço mais interno. Então:

```

for (...)
  for (...) {
    ...
    if (desastre)
      goto erro;
  }
...
erro:
  /* arruma a casa */

```

Essa organização é cômoda se o código para manipulação de erro não for trivial, e se erros puderem ocorrer em vários lugares. Um rótulo tem a mesma forma de um nome de variável, e é seguido por dois pontos $::$. Ele pode aparecer antes de qualquer comando na mesma função onde o goto apareceu.

Um outro exemplo. Considere o problema de encontrar o primeiro elemento negativo num arranjo de duas dimensões. (Arranjos multidimensionais são discutidos no Capítulo 5.) Uma possibilidade é:

```

for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    if (v[i][j] < 0)
      goto achei;
/* nao achei */
...
achei:
  /* achei na posicao i, j */
...

```

Código envolvendo um goto sempre pode ser escrito sem o mesmo, embora talvez com o custo de alguns testes repetidos ou de uma variável adicional. Por exemplo, a pesquisa do arranjo ficaria:

```

achei = 0;
for (i = 0; i < N && !achei; i++)
  for (j = 0; j < M && !achei; j++)
    achei = v[i][j] < 0;
if (achei)
  /* achei na posicao i - 1, j - 1 */
...
else
  /* nao achei */
...

```

Embora nós não sejamos dogmáticos no assunto, parece-nos que o goto deva ser usado raramente, ou nunca.

Capítulo 4

FUNÇÕES E ESTRUTURA DE UM PROGRAMA

Funções dividem grandes tarefas de computação em tarefas menores, e permitem às pessoas trabalharem sobre o que outras já fizeram, ao invés de partir do nada. Funções apropriadas podem frequentemente esconder detalhes de operação de partes de programa que não necessitam conhecê-las, esclarecendo o todo, e facilitando as mudanças.

C foi projetada com funções eficientes e fáceis de usar; programas em C geralmente consistem de várias pequenas funções ao invés de poucas de maior tamanho. Um programa pode residir em um ou mais arquivos-fonte de qualquer forma conveniente; os arquivos-fonte podem ser compilados separadamente e carregados juntos, junto com funções de bibliotecas previamente compiladas. Não discutiremos este processo aqui, já que os detalhes variam de acordo com o sistema local.

A maioria dos programadores está familiarizada com funções de biblioteca para entrada e saída (`getchar`, `putchar`) e cálculos numéricos (`sin`, `cos`, `sqrt`). Neste capítulo entraremos em mais detalhes sobre a escrita de novas funções.

4.1 Conceitos Básicos

Para iniciar, vamos projetar e escrever um programa para imprimir cada linha da sua entrada que contenha um “padrão” ou cadeia de caracteres particular. (Isto é um caso especial do utilitário *grep* disponível no UNIX.) Por exemplo, a pesquisa do padrão “an” no conjunto de linhas

O sertanejo é, antes de tudo, um forte.
A sua aparência, entretanto,
ao primeiro lance de vista,
revela o contrário.

produz a seguinte saída

O sertanejo é, antes de tudo, um forte.
A sua aparência, entretanto,
ao primeiro lance de vista,

A estrutura básica da tarefa divide-se facilmente em três partes:

```
while (há outra linha)
    if (a linha contém o padrão)
        imprima-a
```

Embora seja certamente possível colocar o código para tudo isso na rotina `main`, uma so-

lução melhor é a de usar a estrutura natural fazendo de cada parte uma função separada. Três pequenas peças são mais fáceis de manipular que uma única grande, porque detalhes irrelevantes podem ser escondidos nas funções, e porque a chance de introduzir interações indesejadas é minimizada. E as peças podem até ser de utilidade por si só.

“Enquanto há outra linha” é `lelinha`, uma função que nós escrevemos no Capítulo 1, e “imprima-a” é `printf`, que alguém já nos forneceu. Isso significa que temos de escrever apenas uma rotina que decida se a linha contém uma ocorrência do padrão. Podemos resolver este problema roubando um projeto do PL/1: a função `index (s, t)` que retoma a posição ou índice na cadeia `s` onde a cadeia `t` começa ou `- 1` se `s` não contiver `t`. Usamos `0` ao invés de `1` como posição inicial em `s` porque arranjos em `C` começam na posição zero. Quando posteriormente precisarmos de uma pesquisa de padrão mais sofisticada, teremos que mudar apenas `index`; o resto do código pode permanecer inalterado.

Dado o projeto até este ponto, preencher os detalhes do programa é simples. O programa inteiro segue, de modo que você possa ver como as peças se encaixam. Por ora, o padrão a ser pesquisado é uma cadeia literal no argumento para `index`, o que não é um mecanismo muito geral. Retornaremos brevemente à discussão de como inicializar arranjos de caracteres, e, no Capítulo 5, mostraremos como passar o padrão como parâmetro dado quando o programa é executado. Incluímos também uma nova versão de `lelinha`, você poderá achar instrutivo compará-la com a versão do Capítulo 1.

```
#define    MAXLINHA 1000

main ( )    /* imprime as linhas casando um padrão */
{
    char linha [MAXLINHA];

    while (lelinha (linha, MAXLINHA) > 0)
        if (index (linha, "an") >= 0)
            printf ("%s", linha);
}

lelinha (s, lim) /* le a linha em s, retorna o tamanho */
char s [ ];
int lim;
{
    int c, i;

    i = 0;
    while (-- lim > 0 && (c = getchar ( )) != EOF && c != '\n')
        s [i + +] = c;
    if (c == '\n')
        s [i + +] = c;
    s [i] = '\0';
    return (i);
}

index (s, t) /* retorna índice de t em s, - 1 se não tiver
char s [ ], t [ ];
```



```

int i, j, k;
for (i = 0; s[i] != '\0'; i++) {
    for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++);
    if (t[k] == '\0')
        return (i);
}
return (-1);
}

```

Cada função tem a forma

```

nome (lista de argumentos, se houver)
declarações de argumentos, se houver
{
    declarações e comandos, se houver
}

```

Como sugerido acima, as várias partes podem ser omitidas; uma função mínima é

```
vazia () { }
```

que não faz nada. (Uma função vazia é útil às vezes durante o desenvolvimento de um programa.) O nome da função pode também ser precedido por um tipo se a função retorna alguma outra coisa que um inteiro; este é o tópico da próxima seção.

Um programa é então um conjunto de definições individuais de funções. A comunicação entre funções é feita (neste caso) por argumentos e valores retornados pelas funções; e ela poderia ser feita também através de variáveis externas. As funções podem ocorrer em qualquer ordem no arquivo-fonte, e o programa fonte pode ser dividido em arquivos múltiplos, desde que nenhuma função seja dividida.

O comando `return` é o mecanismo para retornar um valor de uma função chamada, para sua chamadora. Qualquer expressão pode seguir um `return`:

```
return (expressão)
```

A função chamadora pode ignorar o valor retornado se o desejar. Além do mais, não é necessário que uma expressão siga o `return`; nesse caso nenhum valor é retornado ao chamador. O controle também retorna ao chamador sem um valor, quando a execução segue até o fim de uma função passando pelo fecho-chave final da mesma. Não é ilegal, mas indica provavelmente um problema, se a função retornar um valor em alguns lugares e em outros não. Em qualquer caso, o "valor" de uma função que não retorna um valor é sempre lixo. O verificador C lint verifica tais erros.

O mecanismo de compilação e carregamento de um programa C que reside em múltiplos arquivos varia de sistema para sistema. No sistema UNIX, por exemplo, o comando `cc` mencionado no Capítulo 1 é usado. Suponha que as três funções estejam em três arquivos chamados *main.c*, *lelinha.c* e *index.c*. Então o comando

```
cc main.c lelinha.c index.c
```

compila os três arquivos, coloca o código objeto relocável resultante nos arquivos *main.o*, *lelinha.o* e *index.o*, e carrega-os num arquivo executável chamado *a.out*.

Se houver um erro em, digamos, *main.c*, este arquivo pode ser recompilado e o resultado carregado com os arquivos objetos anteriores, com o comando

cc main.c lelinha.o index.o

O comando *cc* usa os sufixos *“.c”* e *“.o”* para diferenciar arquivos-fonte de objetos.

Exercício 4-1. Escreva a função *rindex* (*s*, *t*) que retorna a posição da ocorrência *mais à direita* de *t* em *s*, ou *-1* se *t* não ocorrer em *s*.

4.2 Funções que Retornam Valores não Inteiros

Até agora, nenhum de nossos programas conteve qualquer declaração do tipo de uma função. Isto ocorreu porque, por default, uma função é implicitamente declarada com sua ocorrência numa expressão ou comando, tal como

```
while (lelinha (linha, MAXLINHA) > 0)
```

Se um nome que não foi previamente declarado ocorre em uma expressão e é seguido por um parêntese esquerdo, ele é declarado pelo contexto como sendo o nome de uma função. Além do mais, por default, presume-se que a função retorne um *int*. Visto que *char* é promovido a *int* em expressões, não há necessidade de declarar funções que retornam *char*. Essas regras cobrem a maioria dos casos, incluindo todos os nossos exemplos até agora.

Mas o que acontece se uma função precisa retornar algum outro tipo? Muitas funções numéricas do tipo *sqrt*, *sin*, e *cos* retornam *double*; outras funções especializadas retornam outros tipos. Para ilustrar como lidar com isto, vamos escrever e usar a função *atof* (*s*) que converte a cadeia *s* para ponto flutuante no seu equivalente em ponto flutuante de precisão dupla. *atof* é uma extensão de *atoi*, da qual escrevemos várias versões nos Capítulos 2 e 3; ela aceita um sinal e um ponto decimal opcionais, e a presença ou ausência da parte inteira ou fracionária. (Esta rotina de conversão de entrada *não* é de alta qualidade; tal rotina tomaria mais espaço do que queremos usar.)

Primeiro, *atof* deve declarar o tipo do valor que retorna, desde que ele não é *int*. Visto que *float* é convertido para *double* em expressões, não tem porque dizer que *atof* retorna *float*; é melhor aproveitar a precisão dupla, e declará-la então como retornando *double*. O nome do tipo precede o nome da função, como mostrado a seguir:

```
double atof (s) /* converte a cadeia s a um double */
char s [ ];
{
    double val, pot;
    int i, sinal;

    for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
        /* pula espaco branco */
    sinal = 1;
    if (s[i] == '+' || s[i] == '-') /* sinal */
        sinal = (s[i++] == '+') ? 1 : -1;
    for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + s[i] - '0';
    if (s[i] == '.')
        i++;
    for (pot = 1; s[i] >= '0' && s[i] <= '9'; i++) {
```

```

    val = 10 * val + s[i] - '0';
    pot *= 10;
}
return (sinal * val / pot);
}

```

Segundo, e de igual importância, a rotina *chamadora* deve estabelecer que *atof* retorna um valor não inteiro. A declaração é mostrada na seguinte calculadora de mesa primitiva (adequada apenas para calcular o saldo no talão de cheques), que lê um número por linha, opcionalmente precedido por um sinal, e soma tudo, imprimindo o resultado após cada entrada.

```

#define    MAXLINHA 1000

main ( )    /* calculadora elementar */
{
    double soma, atof ( );
    char linha [MAXLINHA];

    soma = 0;
    while (strlen (linha, MAXLINHA) > 0)
        printf ("%t%.2f\n", soma += atof (linha));
}

```

A declaração

```
double soma, atof ( );
```

indica que *soma* é uma variável do tipo *double*, e que *atof* é uma função retornando um valor *double*. Como mnemônico, ela sugere que ambas *soma* e *atof ()* são valores em ponto flutuante de precisão dupla.

A menos que *atof* seja explicitamente declarada em ambos os lugares, C assume que ela retorna um inteiro, e você obterá resultados sem sentido. Se *atof* em si e sua chamada no *main* são de tipos inconsistentes no mesmo arquivo-fonte, isso será detectado pelo compilador. Mas se (como é mais comum) *atof* tivesse sido compilada separadamente, a inconsistência não seria detectada, e *atof* retornaria um *double* que *main* trataria como *int*, e respostas sem sentido seriam obtidas (*lint* detecta este erro).

Dada *atof*, poderíamos em princípio escrever *atoi* (converte uma cadeia para *int*) em termos de *atof*:

```

atoi (s) /* converte s a um inteiro */
char s [ ];
{
    double atof ( );

    return (atof (s));
}

```

Observe a estrutura das declarações e o comando *return*. O valor da expressão em *return (expressão)*

é sempre convertida para o tipo da função antes do retorno ser feito. Dessa forma, o valor de `atof`, um `double`, é convertido automaticamente para `int` quando ele aparece num `return`, já que a função `atoi` retorna um `int`. (A conversão de um valor em ponto flutuante para `int` trunca qualquer parte fracionária, como discutido no Capítulo 2.)

Exercício 4-2. Estenda `atof` de modo que ela aceite notação científica da forma `123.45e-6` onde um número em ponto flutuante pode ser seguido por `e` ou `E` e um expoente com sinal opcional.

4.3 Mais sobre Argumentos de Funções

No Capítulo 1 nós discutimos o fato de que argumentos de funções são passados por valor, isto é, a função chamada recebe uma cópia temporária e privada de cada argumento, e não seu endereço. Isto significa que a função não pode afetar o argumento original na função chamadora. Dentro de uma função, cada argumento é, essencialmente, uma variável local inicializada com o valor com o qual a função foi chamada.

Quando um nome de arranjo aparece como um argumento para uma função, o endereço do início do arranjo é passado; os elementos não são copiados. A função pode alterar os elementos do arranjo com subíndices a partir dessa posição. O efeito é o da passagem de arranjos por referência. No Capítulo 5, discutiremos o uso de apontadores para permitir que funções afetem variáveis outras que arranjos nas funções chamadoras.

Vale salientar que não há uma maneira inteiramente satisfatória de se escrever uma função transportável que aceite um número variável de argumentos, porque não há um modo transportável da função chamada determinar quantos argumentos foram passados em uma dada chamada. Portanto, não se pode escrever uma função verdadeiramente transportável que determine o máximo entre um número arbitrário de argumentos, como a função embutida `MAX` do Fortran e do PL/1.

É geralmente seguro usar um número variável de argumentos se a função chamada não usar um argumento que não tenha sido fornecido, e se os tipos forem consistentes. `printf`, a função mais comum em C com um número variável de argumentos, usa a informação contida no seu primeiro argumento para determinar quantos argumentos estão presentes e quais são seus tipos. Ela falha desastrosamente se o chamador não fornecer argumentos suficientes ou se os tipos não forem como especificados no primeiro argumento. Ela não é transportável e deve ser modificada para ambientes diferentes.

Alternativamente, se os argumentos são de tipos conhecidos é possível marcar o fim da lista de argumentos de alguma forma especial, tal como um valor especial de argumento (frequentemente zero) marcando o fim destes.

4.4 Variáveis Externas

Um programa C consiste de um conjunto de objetos externos, que são ou variáveis ou funções. O adjetivo “externo” é usado principalmente em contraste com “interno” que descreve os argumentos e as variáveis automáticas definidos dentro das funções. Variáveis externas são definidas fora de qualquer função, e são potencialmente disponíveis para muitas funções. Funções, por sua vez, são sempre externas, porque C não permite que elas sejam definidas dentro de outras funções. Por default, variáveis externas são também “globais”, de modo que toda referência a tais variáveis pelo mesmo nome (mesmo a partir de funções compiladas separadamente) são referências à mesma coisa. Neste

sentido, variáveis externas são análogas ao COMMON do Fortran ou ao EXTERNAL do PL/I. Veremos posteriormente como definir variáveis externas e funções que não são disponíveis globalmente, mas são visíveis somente num único arquivo-fonte.

Visto que as variáveis externas são acessíveis globalmente, elas fornecem uma alternativa para argumentos de funções e valores de retorno na comunicação de dados entre funções. Qualquer função pode acessar uma variável externa pela referência ao seu nome se este tiver sido declarado de alguma forma.

Se um número grande de variáveis devem ser compartilhadas entre funções, variáveis externas são mais convenientes e eficientes que longas listas de argumentos. Como indicado no Capítulo 1, entretanto, este raciocínio deve ser aplicado com alguma cautela, porque ele pode ter um efeito negativo na estrutura de programas, e levar a programas com muitas conexões de dados entre funções.

Uma segunda razão para usar variáveis externas diz respeito à inicialização. Em particular, arranjos externos podem ser inicializados, mas arranjos automáticos não. Falaremos sobre inicialização no final deste capítulo.

A terceira razão para usar variáveis externas é o seu escopo e o seu tempo de vida. Variáveis automáticas são internas a funções; elas passam a existir quando a rotina é iniciada, e desaparecem quando ela termina. Variáveis externas, por outro lado, são permanentes. Elas não vêm e vão, de modo que retêm seus valores de uma ativação de função para outra. Então, se duas funções compartilham algum dado e nenhuma chama a outra, é mais conveniente compartilhar o dado via variáveis externas ao invés de passagem de argumentos.

Vamos examinar estes pontos num exemplo maior. O problema é escrever um outro programa calculador, melhor que o anterior. Este permite +, -, *, /, e = (para imprimir a resposta). Por ser um pouco mais fácil de implementar, a calculadora usará a notação polonesa ao invés da infixada (o esquema polonês reverso é usado por exemplo, pelas calculadoras de bolso da Hewlett-Packard). Na notação polonesa reversa, cada operador segue seus operandos; uma expressão infixada tal como

$$(1 - 2) * (4 + 5) =$$

é fornecida como

$$1\ 2\ -\ 4\ 5\ +\ *\ =$$

Parênteses não são necessários.

A implementação é muito simples. Cada operando é empilhado numa pilha; quando um operador aparece, o número apropriado de operandos (dois para operadores binários) é desempilhado, o operador é aplicado aos mesmos, e o resultado empilhado. No exemplo acima, 1 e 2 são empilhados, e então trocados pela sua diferença, - 1. Depois, 4 e 5 são empilhados e então trocados pela sua soma, 9. O produto de - 1 e 9, que é - 9, os substitui na pilha. O operador = imprime o elemento do topo da pilha sem removê-lo (para que passos intermediários no cálculo possam ser verificados).

As operações de empilhar e desempilhar são triviais, mas, uma vez incluídas a detecção e recuperação de erros, são grandes o suficiente para que seja melhor colocar cada uma numa função separada do que repetir o código ao longo do programa. Deve haver também uma função separada para buscar o próximo operador ou operando da entrada. Então, a estrutura do programa é:

```

while (proximo operador/operando nao e o fim do arquivo)
    if (numero)
        empilha-o
    else if (operador)
        desempilha os operandos
        executa a operacao
        empilha o resultado
    else
        erro

```

A principal decisão do projeto que ainda não foi discutida é onde colocar a pilha, isto é, que rotinas irão acessá-la diretamente. Uma possibilidade é mantê-la em *main*, e passar a pilha e a posição corrente da mesma para as rotinas que empilham e desempilham. Mas *main* não necessita conhecer as variáveis que controlam a pilha; ela deve pensar apenas em termos de empilhamento e desempilhamento. Assim, decidimos declarar a pilha e sua informação associada como variáveis externas acessíveis às funções *empil* e *desempil* mas não a *main*.

A tradução deste esboço em código é fácil. O programa principal é basicamente um grande *switch* no tipo de operador ou operando; isto é, talvez, um uso mais típico do *switch* que o visto no Capítulo 3.

```

#define MAXOP 20 /* tam. max de um operando, operador */
#define NUMERO '0' /* sinal que um numero foi lido */
#define MUITOGRANDE '9' /* sinal que cadeia esta muito grande */

main ( ) /* calculadora polonesa */
{
    int tipo;
    char s [MAXOP];
    double op2, atof ( ), desempil ( ), empil ( );

    while ((tipo = obtemop (s, MAXOP)) != EOF)
        switch (tipo) {
            case NUMERO:
                empil (atof (s));
                break;
            case '+':
                empil (desempil ( ) + desempil ( ));
                break;
            case '*':
                empil (desempil ( ) * desempil ( ));
                break;
            case '-':
                op2 = desempil ( );
                empil (desempil ( ) - op2);
                break;
            case '/':
                op2 = desempil ( );
                if (op2 != 0.0)
                    empil (desempil ( ) / op2);

```

```

        else
            printf ("desempilhou divisor zero\n");
            break;
        case '=':
            printf ("\t%f\n", empil (desempil ( )));
            break;
        case 'z':
            zera ( );
            break;
        case MUITOGRANDE:
            printf ("%20s . . . muito grande\n", s);
            break;
        default:
            printf ("comando %c desconhecido\n", tipo);
            break;
    }
}

#define    MAXVAL 100 /* tamanho maximo da pilha */

int ap = 0;    /* apontador de pilha */
double val[MAXVAL];    /* pilha de valores */
double empil(f)    /* empilha f na pilha de valores */
double f;
{
    if (ap < MAXVAL)
        return (val [ap + +] = f);
    else {
        printf ("erro: pilha cheia\n");
        zera ( );
        return (0);
    }
}

double desempil ( )    /* desempilha o topo da pilha */
{
    if (ap > 0)
        return (val [ -- ap] );
    else {
        printf ("erro: pilha vazia\n");
        zera ( );
        return (0);
    }
}

zera ( ) /* zera a pilha */
{
    ap = 0;
}

```

O comando `z` limpa a pilha com a função `zera`, também usada por `empil` e `desempil` em caso de erro. Discutiremos obtemos brevemente.

Como discutido no Capítulo 1, uma variável é externa se ela é definida fora do corpo de qualquer função. Então a pilha e seu apontador que devem ser compartilhados por `empil`, `desempil`, e `zera` são definidas fora dessas três funções. Mas `main` não faz referência à pilha ou ao seu apontador — a representação é cuidadosamente escondida. Então o código para o operador `=` deve usar

```
empil (desempil ( ));
```

para examinar o topo da pilha sem alterá-lo.

Observe também que, visto que `+` e `*` são operadores comutativos, a ordem em que os operandos são desempilhados é irrelevante, mas para os operadores `-` e `/`, os operadores esquerdo e direito devem ser diferenciados.

Exercício 4-3. Dado o esqueleto básico, é fácil estender a calculadora. Acrescente os operadores módulo (`%`) e menos unário. Acrescente um comando “apaga” que apaga o topo da pilha. Acrescente comandos para a manipulação de variáveis (é fácil introduzir 26 variáveis com nomes sendo uma letra única).

4.5 Regras de Escopo

As funções e variáveis externas que compõem um programa C não necessitam ser compiladas ao mesmo tempo; o texto fonte do programa pode ser colocado em vários arquivos, e rotinas compiladas previamente podem ser carregadas a partir de bibliotecas. As duas questões de interesse são

- Como escrever as declarações de modo que as variáveis sejam declaradas apropriadamente durante a compilação?
- Como escrever declarações de forma a juntar os pedaços apropriadamente quando o programa é carregado?

O *escopo* de um nome é a parte do programa para a qual o nome é definido. Para uma variável automática declarada no início de uma função, o escopo é a função em que o nome é declarado, e variáveis com o mesmo nome em diferentes funções não têm relação entre si. O mesmo é válido para argumentos de funções.

O escopo de uma variável externa vai do ponto em que é declarada no arquivo-fonte até o fim do mesmo. Por exemplo, se `val`, `ap`, `empil`, `desempil`, e `zera` são definidos num arquivo, na ordem mostrada acima, isto é,

```
int ap = 0;
double val [MAXVAL];

double Emil (f) { ... }
double desempil ( ) { ... }
zera ( ) { ... }
```

então as variáveis `val` e `ap` podem ser usadas em `empil`, `desempil` e `zera`, simplesmente pelo nome; novas declarações não são necessárias.

Por outro lado, se uma variável externa deve ser referenciada antes de sua definição, ou se ela é definida em um arquivo-fonte diferente de onde é usada, a declaração `extern` é obrigatória.

É importante distinguir entre a *declaração* e a *definição* de uma variável externa. Uma declaração anuncia as propriedades de uma variável (seu tipo, tamanho, etc.); uma definição também provoca a alocação de memória. Se as linhas

```
int ap;  
double val [MAXVAL];
```

aparecem fora de qualquer função, *definem* as variáveis externas *ap* e *val*, provocando a alocação de memória e também as declaram para o resto do arquivo-fonte. Por outro lado, as linhas

```
extern int ap;  
extern double val [ ];
```

declaram para o resto do arquivo-fonte que *ap* é um *int* e *val* é um arranjo do tipo *double* (cujo tamanho é determinado em algum outro lugar), mas não criam variáveis, nem alocam memória para as mesmas.

Deve haver somente uma *definição* de uma variável externa em todos os arquivos que compõem o programa fonte; outros arquivos podem conter declarações *extern* para acessá-la. (Pode haver também uma declaração *extern* no arquivo contendo a definição.) Qualquer inicialização de uma variável externa vem somente com a definição. Tamanhos de arranjos devem ser especificados na definição, mas são opcionais numa declaração *extern*.

Embora não seja uma organização comum para este programa, *val* e *ap* poderiam ser definidas e inicializadas num arquivo, e as funções *empil*, *desempil*, e *zera* definidas em outro. Então essas definições e declarações seriam necessárias para ligá-las:

No arquivo 1:

```
int ap = 0; /* apontador de pilha */  
double val [MAXVAL]; /* pilha de valores */
```

No arquivo 2:

```
extern int ap;  
extern double val [ ];  
  
double empil (f) { . . . }  
  
double desempil ( ) { . . . }  
  
zera ( ) { . . . }
```

Devido às declarações *extern* no *arquivo 2* aparecerem antes e fora das três funções, elas se aplicam às três; um conjunto de declarações é suficiente para todo o *arquivo 2*.

Para programas maiores, o recurso de inclusão de arquivos incluído posteriormente nesse capítulo permite manter uma única cópia das declarações *extern* para o programa; o arquivo pode ser inserido em cada arquivo-fonte durante sua compilação.

Vamos agora voltar à implementação de *obtemop*, a função que busca o próximo operando ou operador. A tarefa básica é simples: saltar brancos, caracteres de tabulação e de nova-linha. Se o próximo caractere não é um dígito ou um ponto decimal, retorná-lo.

Caso contrário, colocá-lo numa cadeia de dígitos (que pode incluir um ponto decimal), e retornar NUMERO, o sinal de que um número foi encontrado.

A rotina é substancialmente complicada por uma tentativa de manipular apropriadamente a situação da leitura de um número muito grande. obtemop lê dígitos (talvez com um ponto decimal) até não encontrar mais nenhum, mas armazena somente aqueles que couberem. Se não há transbordo, ela retorna NUMERO e a cadeia de dígitos. Se o número for muito grande, entretanto, obtemop desconsidera o resto da entrada de modo que o usuário possa simplesmente redigitar a linha a partir do ponto de erro; ela retorna MUITOGRANDE como sinal de transbordo.

```
obtemop (s, lim) /* obtém próximo operando ou operador */
char s [ ];
int lim;
{
    int i, c;

    while ((c = getch ( )) == ' ' || c == '\t' || c == '\n')
        ;
    if (c != '.' && (c < '0' || c > '9'))
        return (c);
    s[0] = c;
    for (i = 1; (c = getch ( )) >= '0' && c <= '9'; i++)
        if (i < lim)
            s[i] = c;
    if (c == '.') { /* pega a fração */
        if (i < lim)
            s[i] = c;
        for (i++; (c = getch ( )) >= '0' && c <= '9'; i++)
            if (i < lim)
                s[i] = c;
    }
    if (i < lim) { /* número está ok */
        ungetch (c);
        s[i] = '\0';
        return (NUMERO);
    } else { /* número muito grande; pula a linha */
        while (c != '\n' && c != EOF)
            c = getch ( );
        s[lim-1] = '\0';
        return (MUITOGRANDE);
    }
}
```

O que são getch e ungetch? É freqüente o caso em que um programa lendo sua entrada não pode determinar se leu o suficiente até que tenha lido demais. Um exemplo é colecionar caracteres que compõem um número: até o primeiro caractere que não seja um dígito ser visto, o número não está completo. Mas quando ele aparece, o programa leu um caractere a mais, o qual não estava preparado para receber.

O problema seria resolvido se fosse possível “devolver” para a entrada o caractere indesejado. Então, toda vez que o programa lesse um caractere a mais, este poderia ser devolvido à entrada, de modo que o resto do programa pudesse se comportar como se nunca o tivesse visto. Felizmente, é fácil simular isto, escrevendo-se um par de funções cooperantes. `getch` obtém o próximo caractere a ser considerado; `ungetch` devolve um caractere para a entrada, de modo que a próxima chamada a `getch` o retorne.

Seu modo de funcionamento conjunto é simples. `ungetch` coloca o caractere devolvido num buffer compartilhado – um arranjo de caracteres. `getch` lê a partir do buffer, se houver caracteres ali; e chama `getchar` caso contrário. Deve haver também uma variável índice que indique a posição do caractere corrente no buffer.

Desde que o buffer e o índice são compartilhados por `getch` e `ungetch` e devem reter seus valores entre chamadas, eles devem ser externos a ambas as rotinas. Então podemos escrever `getch`, `ungetch`, e suas variáveis compartilhadas como:

```
#define  TAMBUF 100

char buf [TAMBUF]; /* buffer para ungetch */
int abuf = 0; /* proxima posicao livre em buf */

getch ( ) /* obtem o proximo caractere */
{
    return ((abuf > 0) ? buf [ -- abuf ] : getchar ( ));
}

ungetch (c) /* devolve caractere a saida */
int c;
{
    if (abuf >= TAMBUF)
        printf ("ungetch: caracteres demais\n");
    else
        buf [abuf + +] = c;
}
```

Usamos um arranjo para armazenar caracteres devolvidos ao invés de um único caractere, para garantir a generalidade de `getch` e `ungetch` para usos posteriores.

Exercício 4-4. Escreva a rotina `ungets(s)` que devolva uma cadeia inteira para a entrada. `ungets(s)` precisa conhecer `buf` e `bufp` ou precisa usar somente `ungetch`?

Exercício 4-5. Suponha que nunca haverá mais que um caractere a ser devolvido para a entrada. Modifique `getch` e `ungetch` para acomodar esta suposição.

Exercício 4-6. Nossas rotinas `getch` e `ungetch` não manipulam a devolução de EOF de forma transportável. Decida que propriedades devem existir se um EOF é devolvido para a entrada, e implemente seu projeto.

4.6 Variáveis Estáticas

Variáveis estáticas são uma terceira classe de armazenamento, além das variáveis do tipo `extern` e automáticas que já vimos.

Variáveis do tipo `static` podem tanto ser internas como externas. As internas são locais a uma função particular como variáveis automáticas, mas, ao contrário destas, continuam a existir independentemente da ativação ou desativação da função. Isso significa que elas fornecem uma forma de armazenamento privado e permanente em funções. Cadeias de caracteres que aparecem numa função, tais como argumentos de `printf`, são estáticas internas.

Uma variável estática externa é conhecida no resto do *arquivo-fonte* onde é declarada mas não em qualquer outro arquivo. Elas fornecem um meio de esconder nomes como `buf` e `abuf` na combinação `getch` e `ungetch`, que devem ser externas já que são compartilhadas, mas que deveriam ser invisíveis nos usuários de `getch` e `ungetch`, para não haver possibilidade de conflito. Se as duas rotinas e as duas variáveis são compiladas num mesmo arquivo, como

```
static char buf [TAMBUF];
static int abuf = 0;

getch ( ) { ... }

ungetch (c) { ... }
```

então nenhuma outra rotina será capaz de acessar `buf` e `abuf`; de fato, elas não conflitarão com os mesmos nomes em outros arquivos do mesmo programa.

Armazenamento estático interno ou externo, é especificado pelo prefixo `static` numa declaração normal. A variável é externa se é definida fora de qualquer função e interna caso contrário.

Normalmente, funções são objetos externos; seus nomes são conhecidos globalmente. É possível, entretanto, declarar uma função como sendo estática; isto a torna desconhecida fora do arquivo em que foi declarada.

Em C, [`static`] indica não apenas permanência mas também um grau de “privacidade”. Objetos estáticos internos são conhecidos somente em uma função; objetos estáticos externos (variáveis ou funções) são conhecidos somente no arquivo-fonte onde aparecem, e seus nomes não interferem com variáveis ou funções com o mesmo nome em outros arquivos.

Variáveis e funções externas estáticas fornecem um modo de esconder objetos de dados e rotinas internas que as manipulam, de modo que outras rotinas e dados não possam conflitar, mesmo inadvertidamente. Por exemplo, `getch` e `ungetch` formam um “módulo” para a entrada e devolução de caracteres e `buf` e `abuf` devem ser estáticas para que sejam inacessíveis externamente. Do mesmo modo, `empil`, `desempil`, e `zera` formam um módulo para a manipulação de pilha e `val` e `ap` devem ser variáveis estáticas externas.

4.7 Variáveis em Registradores

A quarta e última classe de armazenamento é chamada *register*. Uma declaração *register* avisa ao compilador que a variável em questão será largamente usada. Quando possível, tais variáveis são colocadas em registradores da máquina, o que pode resultar em programas menores e mais rápidos.

A declaração *register* tem o seguinte formato:

```
register int x;
register char c;
```

e assim por diante; a parte `int` pode ser omitida. `register` pode ser aplicado a variáveis automáticas e a parâmetros formais de uma função. No último caso, a declaração tem o formato:

```
f (c, n)
register int c, n;
{
    register int i;
    ...
}
```

Na prática há algumas restrições sobre variáveis em registradores, refletindo a realidade do hardware de suporte. Somente poucas variáveis em cada função podem ser guardadas em registradores, e somente certos tipos são permitidos. A palavra `register` é ignorada no caso de excesso de declarações e declarações não permitidas. E não é possível se obter o endereço de uma variável em registrador (um tópico a ser coberto no Capítulo 5). As restrições específicas variam de máquina para máquina; como exemplo, no PDP-11 somente as três primeiras declarações de registradores são efetivadas, e os tipos podem ser `int`, `char`, ou apontador.

4.8 Estrutura de Bloco

C não é uma linguagem com estrutura de bloco no sentido de PL/I ou Algol, desde que funções não podem ser definidas dentro de outras funções.

Por outro lado, variáveis podem ser definidas seguindo uma estrutura de bloco. Declarações de variáveis (incluindo inicializações) podem seguir a abre-chaves de *qualquer* comando composto, e não somente após a chave que inicia uma função. Variáveis declaradas desta forma escondem quaisquer variáveis com nomes idênticos em outros blocos, e permanecem em existência até o fecha-chaves correspondente. Por exemplo, em

```
if (n > 0) {
    int i; /* declara um novo i */
    for (i = 0; i < n; i++)
        ...
}
```

o escopo da variável `i` é o desvio verdadeiro do `if`; este `i` não tem nenhuma relação com qualquer outro `i` do programa.

A estrutura de bloco também se aplica a variáveis externas. Dadas as declarações

```
int x;

f ()
{
    double x;
    ...
}
```

dentro da função `f`, ocorrências de `x` referem-se à variável interna do tipo `double`; fora de `f`, referem-se ao inteiro externo. O mesmo é válido para nomes de parâmetros formais:

```

int z;
f (z)
double z;
{
    ...
}

```

Dentro da função *f*, *z* refere-se ao parâmetro formal, e não à variável externa.

4.9 Inicialização

Inicialização tem sido mencionada de passagem por muitas vezes até agora, mas sempre periféricamente quando da referência a outro tópico. A presente seção sumariza algumas das regras, uma vez que já discutimos as várias classes de armazenamento.

Na falta de inicialização explícita, variáveis externas e variáveis estáticas são inicializadas com o valor zero; variáveis automáticas e em registradores têm valor indefinido (i.e., lixo).

Variáveis simples (não arranjos e estruturas) podem ser inicializadas quando são declaradas, seguindo o nome com um sinal de igualdade e uma expressão constante:

```

int x = 1;
char apostrofo = '\';
long dia = 60 * 24; /* minutos em um dia */

```

Para variáveis externas e variáveis estáticas, a inicialização é feita uma vez, conceitualmente quando da compilação. Para variáveis automáticas e em registradores, a inicialização é feita cada vez que a função ou bloco é iniciado.

Para variáveis automáticas e em registradores, o inicializador não se restringe a uma constante: ele pode ser qualquer expressão válida envolvendo valores previamente definidos, até mesmo chamadas de funções. Por exemplo, as inicializações do programa de pesquisa binária do Capítulo 3 poderiam ser escritas como:

```

pesq_binaria (x, v, n)
int x, v [ ], n;
{
    int inicio = 0;
    int fim = n - 1;
    int meio;
    ...
}

```

ao invés de

```

pesq_binaria (x, v, n)
int x, v [ ], n;
{
    int inicio, fim, meio;

    inicio = 0;
    fim = n - 1;
    ...
}

```

Efetivamente, inicializações de variáveis automáticas são uma forma de abreviar comandos de atribuição. Que forma de inicialização usar é uma questão de gosto. Usamos geralmente a atribuição explícita, porque inicializações em declarações são mais difíceis de se ver.

Arranjos automáticos não podem ser inicializados. Arranjos externos e estáticos podem ser inicializados seguindo-se a declaração com uma lista de inicializadores entre chaves e separados por vírgulas. Por exemplo, o programa de contagem de caracteres do Capítulo 1, que começava

```
main ( ) /* conta dígitos, espaço branco, outros */
{
    int c, i, nbranco, noutro;
    int ndigito [10];

    nbranco = noutro = 0;
    for (i = 0; i < 10; ++ i)
        ndigito [i] = 0;
    ...
}
```

poderia ser escrito como

```
int nbranco = 0;
int noutro = 0;
int ndigito [10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
main ( ) /* conta dígitos, espaço branco, outros */
{
    int c, i;
    ...
}
```

Na realidade, essas inicializações são desnecessárias desde que são todas iguais a zero, mas é uma boa forma de torná-las explícitas. Se há menos inicializadores que o tamanho especificado, os outros serão zero. É um erro ter-se mais inicializadores que o necessário. Lamentavelmente não há como se especificar a repetição de um inicializador, nem de se inicializar um elemento no meio do arranjo sem inicializar todos os elementos anteriores ao mesmo.

Arranjos de caracteres são um caso especial de inicialização; uma cadeia pode ser usada ao invés de chaves e vírgulas:

```
char padrao [ ] = "uma";
```

Isto é uma abreviação para a seguinte construção mais extensa:

```
char padrao [ ] = { 'u', 'm', 'a', '\0' };
```

Quando o tamanho de um arranjo de qualquer tipo é omitido o compilador calcula o tamanho contando os inicializadores. Neste caso específico, o tamanho é 4 (três caracteres mais o terminador \0).

4.10 Recursividade

Funções C podem ser usadas recursivamente; isto é, uma função pode chamar a si própria direta ou indiretamente. Um exemplo tradicional é a impressão de um número

como uma cadeia de caracteres. Como mencionamos antes, os dígitos são gerados na ordem inversa: dígitos de mais baixa ordem estão disponíveis antes que os de mais alta ordem, mas eles devem ser impressos na ordem inversa.

Existem duas soluções para este problema. Uma é a de armazenar os dígitos num arranjo quando eles são gerados e então imprimi-los na ordem inversa, como fizemos no Capítulo 3 com itoa. A primeira versão de imprdec segue esse padrão.

```
imprdec (n) /* imprime n em decimal */
int n;
{
    char s[10];
    int i;

    if (n < 0) {
        putchar ('-');
        n = -n;
    }
    i = 0;
    do {
        s[i++] = n % 10 + '0'; /* obtem proximo caractere */
    } while ((n /= 10) > 0); /* jogue-o fora */
    while (--i >= 0)
        putchar (s[i]);
}
```

A alternativa é usar recursividade, onde cada chamada a imprdec primeiro chama a si própria para imprimir os dígitos iniciais, e então imprime o dígito final.

```
imprdec (n) /* imprime n em decimal (recursivo) */
int n;
{
    int i;

    if (n < 0) {
        putchar ('-');
        n = -n;
    }
    if ((i = n/10) != 0)
        imprdec (i);
    putchar (n % 10 + '0');
}
```

Quando uma função chama a si própria recursivamente, cada ativação recebe um conjunto novo de todas as variáveis locais, independente do conjunto anterior. Então, em imprdec (123) o primeiro imprdec tem $n = 123$. Ele passa 12 para o segundo imprdec, e depois imprime 3 quando este segundo imprdec retornar. Do mesmo modo o segundo imprdec passa 1 para o terceiro (que o imprime) e então imprime 2.

Recursividade em geral não fornece economia de memória, desde que em algum lugar uma pilha de valores sendo processados tem de ser montada. Nem será mais rápida. Mas o código recursivo é mais compacto, e freqüentemente mais fácil de escrever e enten-

der. Recursividade é especialmente conveniente para estrutura de dados definidas recursivamente tais como de árvores; veremos um bom exemplo no Capítulo 6.

Exercício 4-7. Adapte a idéia de `imprdec` para escrever uma versão recursiva de `itoa`; isto é, converter um inteiro em uma cadeia com uma rotina recursiva.

Exercício 4-8. Escreva uma versão recursiva da função `inverte (s)`, que inverte a cadeia `s`.

4.11 O Preprocessador C

C fornece certas extensões de linguagem por meio de um simples macroprocessador. A facilidade `#define` que temos usado é a mais comum dessas extensões; uma outra é a habilidade de incluir o conteúdo de outros arquivos durante a compilação.

Inclusão de Arquivos

Para simplificar o manuseio de coleções de `#define's` e declarações (entre outras coisas), C fornece a facilidade de inclusão de arquivo. Qualquer linha parecida com

```
#include "arquivo"
```

é substituída pelo conteúdo do arquivo chamado *arquivo*. (As aspas são obrigatórias.) Frequentemente, uma ou duas linhas desta forma aparecem no início de cada arquivo-fonte, para incluir comandos `#define` comuns e declarações do tipo `extern` para variáveis globais. `#define's` podem estar aninhados.

`#include` é a forma preferida de se juntar declarações para um grande programa. Ela garante que todos os arquivos-fonte serão supridos com as mesmas definições e declarações de variáveis e elimina assim um tipo de erro particularmente desagradável. Evidentemente, quando um arquivo de inclusão é alterado, todos os arquivos que dependem dele devem ser recompilados.

Substituição de Macros

Uma definição da forma

```
#define SIM 1
```

especifica uma substituição de macro do tipo mais simples – substituindo um nome por uma cadeia de caracteres. Nomes num `#define` têm a mesma forma de um identificador C; o texto de reposição é arbitrário. Normalmente, o texto de reposição é o resto da linha; uma definição longa pode ser continuada na outra linha colocando-se um `\` no final da linha a ser continuada. O “escopo” de um nome definido com `#define` vai do ponto da definição até o fim do arquivo-fonte. Nomes podem ser redefinidos, e uma definição pode usar definições anteriores. Substituições não são feitas em cadeias de caracteres, assim, por exemplo, se `SIM` é um nome definido não haverá substituição em `printf (“SIM”)`.

Desde que a implementação de `#define` é feita com um passo de preprocessamento que não faz parte integrante do compilador, há muito poucas restrições gramaticais no que pode ser definido. Por exemplo, os aficionados de Algol podem fazer

```
#define then  
#define begin {  
#define end ; }
```

e escrever

```
if (i > 0) then
  begin
    a = 1;
    b = 2
  end
```

É também possível definir macros com argumentos, de forma que o texto de reposição dependa da forma como a macro é chamada. Como exemplo, definamos uma macro chamada `max` como segue:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Agora a linha

```
x = max (p + q, r + s);
```

será substituída por

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Isto fornece uma “função máximo” que se expande numa linha de código ao invés de uma chamada de função. Além disso, como os argumentos são tratados de forma consistente, esta macro servirá para qualquer tipo de dados; não há necessidade de definirmos diferentes tipos de `max` para diferentes tipos de dados, como seria necessário com funções.

É claro, se você examinar a expressão acima, notará algumas armadilhas. As expressões são avaliadas duas vezes; isto é ruim se elas envolvem efeitos colaterais tais como chamadas de funções e operadores de incremento. Algum cuidado deve ser tomado com os parênteses, para garantir a ordem de avaliação. (Considere a macro

```
#define quadrado(x) x * x
```

quando invocada com `quadrado(z + 1)`.) Existem algumas restrições léxicas também: não pode haver espaço entre o nome e o parêntese esquerdo que introduz a lista de argumentos.

No entanto, macros são muito valiosas. Um exemplo prático é a biblioteca-padrão de entrada e saída descrita no Capítulo 7, em que `getchar` e `putchar` são definidos como macros (obviamente `putchar` necessita de um argumento), evitando desta forma o desperdício de uma chamada de função a cada caractere processado.

Outras facilidades do macroprocessador são descritas no Apêndice A.

Exercício 4-9. Defina uma macro `troca(x, y)` que troque o valor de dois argumentos inteiros. (A estrutura de bloco ajudará.)

Capítulo 5

APONTADORES E ARRANJOS

Um apontador é uma variável que contém o endereço de outra variável. Apontadores são muito usados em C, em parte porque eles são, às vezes, a única forma de se expressar uma computação, e em parte porque eles normalmente levam a um código mais compacto e eficiente que o obtido de outras formas.

Apontadores têm sido comparados ao comando `goto` como uma forma maravilhosa de se criar programas impossíveis de entender. Isto é certamente verdade quando eles são usados sem cuidado, e é fácil criar apontadores que apontem para algum lugar inesperado. Com disciplina, entretanto, apontadores podem ser usados para se obter clareza e simplicidade. Este é o aspecto que tentaremos ilustrar.

5.1 Apontadores e Endereços

Desde que um apontador contém o endereço de um objeto, é possível acessar o objeto indiretamente através do apontador. Suponha que `x` é uma variável, digamos um `int`, e que `px` é um apontador, criado de alguma forma ainda não especificada.

O operador unário `&` fornece o endereço de um objeto, de forma que o comando

```
px = &x;
```

atribui o endereço de `x` à variável `px`; diz-se então que `px` “aponta” para `x`. O operador `&` pode ser aplicado somente a variáveis e elementos de arranjos; construções tais como `&(x + 1)` e `&3` são ilegais. É ilegal também obter o endereço de uma variável da classe registrador.

O operador unário `*` trata seu operando como um endereço, e acessa este endereço para buscar o conteúdo do objeto alvo. Então se `y` também é um `int`,

```
y = *px;
```

atribui a `y` o conteúdo do objeto para o qual `px` aponta. Assim a sequência

```
px = &x;
```

```
y = *px;
```

atribui a `y` o mesmo valor atribuído no comando

```
y = x;
```

É necessário também declarar todas as variáveis usadas nestas construções

```
int x, y;
```

```
int *px;
```

A declaração de `x` e `y` já nos é familiar. A declaração do apontador `px` é nova. A declaração `int *px;`

é um mnemônico; ela diz que a combinação `*px` é um `int`, isto é, é equivalente a uma variável do tipo `int`. Isto é, a sintaxe da declaração de uma variável tem forma similar à sintaxe das expressões em que a variável pode aparecer. Este raciocínio é útil em todos os casos envolvendo declarações complicadas. Por exemplo,

```
double atof ( ), *dp;
```

diz que em uma expressão, `atof ( )` e `*dp` têm valores do tipo `double`.

Você deve também observar que a declaração implica que um apontador restringe-se a apontar para um tipo particular de objeto.

Apontadores podem ocorrer em expressões. Por exemplo, se `px` aponta para o inteiro `x`, então `*px` pode ocorrer em qualquer contexto em que `x` possa ocorrer.

```
y = *px + 1
```

atribui a `y` o valor de `x` mais um;

```
printf ("%d\n", *px)
```

imprime o valor de `x`; e

```
d = sqrt ((double) *px)
```

atribui a `d` a raiz quadrada de `x`, o qual é convertido em um `double` antes de ser passado para `sqrt`. (Veja o Capítulo 2.)

Em expressões tais como

```
y = *px + 1
```

Os operadores unários `*` e `&` têm precedência maior que os operadores aritméticos, de forma que esta expressão toma o valor do objeto apontado por `px`, adiciona 1 e atribui o valor a `y`. Veremos logo o significado da construção

```
y = *(px + 1)
```

Referências a apontadores podem também ocorrer no lado esquerdo de atribuições. Se `px` aponta para `x`, então

```
*px = 0
```

atribui zero a `x`, e

```
*px += 1
```

incrementa `x`, da mesma forma que

```
(*px) ++
```

Os parênteses são necessários neste último exemplo; sem eles, a expressão incrementaria `px` ao invés do objeto para o qual `px` aponta, porque operadores unários tais como `*` e `++` são avaliados da direita para a esquerda.

Finalmente, desde que apontadores são variáveis, eles podem ser manipulados como tal. Se `py` é um outro apontador para um `int`, então

```
py = px
```

copia o conteúdo de `px` em `py`, de forma que `py` aponte também para o objeto apontado por `px`.

5.2 Apontadores e Argumentos de Funções

Como C passa argumentos para funções usando “chamada por valor”, a função chamada não pode alterar diretamente uma variável na função chamadora. O que devemos fazer se tivermos, realmente, de alterar um argumento normal? Por exemplo, uma rotina de ordenação poderia permutar dois elementos fora de ordem com uma função chamada *troca*. Não é suficiente escrever

```
troca(a, b);
```

onde a função *troca* é definida como

```
troca (x, y) /* ERRADO */
int x, y;
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Por causa da chamada por valor, *troca* não pode afetar os argumentos `a` e `b` na rotina que a chama.

Felizmente, há uma maneira de obter o efeito desejado. O programa chamador passa *apontadores* para os valores a serem permutados:

```
troca (&a, &b);
```

Desde que o operador `&` fornece o endereço de uma variável, `&a` é um apontador para `a`. Em *troca*, os argumentos são declarados como apontadores, e os operandos atuais são acessados através deles.

```
troca (ax, ay) /* permuta *ax e *ay */
int *ax, *ay;
{
    int temp;

    temp = *ax;
    *ax = *ay;
    *ay = temp;
}
```

Um uso comum de argumentos do tipo apontador ocorre em funções que devem retornar mais que um valor. (Pode-se dizer que *troca* retorna dois valores, os novos valores

dos argumentos.) Como exemplo, considere uma função `leint` que faz a conversão de sua entrada em formato livre, separando uma sequência de caracteres em valores inteiros, um inteiro por chamada. `leint` deve retornar o valor encontrado ou um sinalizador de fim de arquivo, quando não houver mais entrada. Esses valores têm de ser retornados em objetos distintos, pois não importa que valor é usado para EOF, este valor poderia ser também o valor de um inteiro na entrada.

Uma solução, baseada na função de entrada `scanf` que será descrita no Capítulo 7, faz com que `leint` retorne o valor EOF, caso seja encontrado o fim do arquivo e qualquer outro valor, caso contrário. O valor numérico do inteiro encontrado é retornado através de um argumento que deve ser um apontador para um inteiro. Esta organização separa o sinalizador de fim de arquivo dos valores numéricos.

O laço seguinte preenche um arranjo com inteiros através de chamadas a `leint`:

```
int n, v, arranjo [TAMANHO];
```

```
for (n = 0; n < TAMANHO && leint (&v) != EOF; n++)
    arranjo [n] = v;
```

Cada chamada atribui a `v` o próximo inteiro encontrado na entrada. Observe que é essencial escrever `&v` ao invés de `v` como argumento de `leint`. O uso de `v` causará provavelmente um erro de endereçamento, já que `leint` acredita estar manipulando um apontador válido.

`leint` é uma modificação óbvia de `atoi` que escrevemos anteriormente:

```
leint (an) /* le próximo inteiro da entrada */
int *an;
{
    int c, sinal;

    while ((c = getch ()) == ' ' || c == '\n' || c == '\t')
        /* pula espaço em branco */
        sinal = 1;
    if (c == '+' || c == '-') { /* guarda o sinal */
        sinal = (c == '+') ? 1 : -1;
        c = getch ();
    }
    for (*an = 0; c >= '0' && c <= '9'; c = getch ())
        *an = 10 * *an + c - '0';
    *an *= sinal;
    if (c != EOF)
        ungetch (c);
    return (c);
}
```

Em `leint`, `*an` é usado como uma variável normal do tipo `int`. Usamos também `getch` e `ungetch` (descritas no Capítulo 4) de modo que um caractere adicional lido possa ser devolvido para a entrada.

Exercício 5-1. Escreva `lefloat`, análoga a `leint` para ponto flutuante. Qual é o tipo do valor retornado por `lefloat`?

5.3 Apontadores e Arranjos

Em C o relacionamento entre apontadores e arranjos é tão estreito que apontadores e arranjos deveriam ser realmente tratados juntos. Qualquer operação que possa ser feita com índices de um arranjo, pode ser feita com apontadores. A versão com apontador será, em geral, mais rápida, mas, pelo menos para os iniciantes, mais difícil de compreender imediatamente.

A declaração

```
int a [10]
```

define um arranjo *a* de tamanho 10, isto é, um bloco de 10 objetos consecutivos chamados *a* [0], *a* [1], ..., *a* [9]. A notação *a* [*i*] significa o elemento da *i*-ésima posição do arranjo a partir do início do mesmo. Se *pa* é um apontador para um inteiro, declarado como segue:

```
int *pa;
```

então a atribuição

```
pa = &a [0]
```

faz com que *pa* aponte para o zero-ésimo elemento de *a*; isto é, *pa* contém o endereço de *a* [0]. Agora a atribuição

```
x = *pa;
```

copiará o conteúdo de *a* [0] em *x*.

Se *pa* aponta para um elemento particular de um arranjo *a*, então por definição *pa* + 1 aponta para o próximo elemento, e em geral *pa* - *i* aponta para *i* elementos antes de *pa*, e *pa* + *i* aponta para *i* elementos após. Então, se *pa* aponta para *a* [0],

```
* (pa + 1)
```

refere-se ao conteúdo de *a* [1], *pa* + *i* é o endereço de *a* [*i*], e ** (pa + i)* é o conteúdo de *a* [*i*].

Estas observações aplicam-se independentemente do tipo das variáveis no arranjo *a*. A definição de "somar um a um apontador", e por extensão, toda a aritmética com apontadores, é a de que o incremento é modificado para refletir o tamanho do objeto apontado. Então em *pa* + *i*, *i* é multiplicado pelo tamanho do objeto apontado por *pa* antes de ser somado a *pa*.

A correspondência entre indexação e aritmética com apontadores é evidentemente muito estreita. De fato, uma referência para um arranjo é convertida pelo compilador a um apontador para o início do arranjo. O efeito é que um nome de arranjo é uma expressão do tipo apontador. Isto tem várias implicações úteis. Desde que o nome de um arranjo é um sinônimo para a posição do elemento de índice 0, a atribuição:

```
pa = &a [0];
```

pode ser escrita como

```
pa = a;
```

Ainda mais surpreendente, pelo menos à primeira vista, é o fato de que uma referência a *a* [*i*] pode ser escrita como ** (a + i)*. Na avaliação da expressão de *a* [*i*], C a converte para ** (a + i)* imediatamente; as duas formas são completamente equivalentes. Pela aplicação do operador *&* a ambas as partes dessa equivalência, segue que *&a* [*i*] e *a* + *i* são

idênticas: $a + i$ é o endereço do i -ésimo elemento após a . Por outro lado, se pa é um apontador, expressões podem usá-lo com um índice: $pa[i]$ é idêntico a $*(pa + i)$. Em suma, qualquer arranjo e expressão indexada podem ser escritos como um apontador e um deslocamento, e vice-versa, até no mesmo comando.

Há uma diferença entre o nome de um arranjo e um apontador que deve ser lembrada. Um apontador é uma variável, de forma que $pa = a$ e $pa++$ são operações possíveis. Mas o nome de um arranjo é uma *constante*, não uma variável: construções do tipo $a = pa$ ou

$a++$

ou

$p = \&a$

são ilegais.

Quando o nome de um arranjo é passado para uma função, o que é passado é a posição do início do arranjo. Dentro da função chamada, este argumento é uma variável, exatamente como outra qualquer, de forma que um argumento do tipo nome de arranjo é verdadeiramente um apontador, isto é, uma variável contendo um endereço. Podemos usar este fato para escrever uma nova versão de `strlen`, que calcula o tamanho de uma cadeia.

```
strlen(s) /* retorna o tamanho da cadeia s */
char *s;
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return (n);
}
```

Incrementar s é perfeitamente legal, desde que s é uma variável do tipo apontador; $s++$ não tem nenhum efeito na cadeia de caracteres na função que chamou `strlen`, mas simplesmente incrementa uma cópia privativa do endereço.

Como parâmetros formais numa definição de função, as construções

```
char s [ ];
```

e

```
char *s;
```

são exatamente equivalentes; a forma a adotar é determinada pela forma de escrever expressões na função. Quando o nome de um arranjo é passado para uma função, a função pode, de acordo com sua conveniência, acreditar que ela está manipulando tanto um arranjo como um apontador, e manipulá-lo de acordo. Ela pode até mesmo usar ambos os tipos de operações se lhe parecer apropriado e claro.

É possível passar parte de um arranjo para uma função, passando um apontador para o início do subarranjo. Por exemplo, se a é um arranjo,

```
f(&a[2])
```

e

```
f(a + 2)
```


passam para a função *f* o endereço do elemento *a* [2] porque *&a* [2] e *a* + 2 são expressões do tipo apontador que se referem ao terceiro elemento de *a*. Em *f*, a declaração do argumento pode ser

```
f (arr)
int arr [ ];
{
    ...
}
```

ou

```
f (arr)
int *arr;
{
    ...
}
```

Assim, do ponto de vista da função *f*, o fato de que o argumento se refere realmente a uma parte de um arranjo maior não traz nenhuma consequência.

5.4 Aritmética com Endereços

Se *p* é um apontador, então *p* ++ incrementa *p* para apontar para o próximo elemento, qualquer que seja o tipo de objeto para o qual *p* aponta, e *p* += *i* incrementa *p* para apontar *i* elementos além do objeto para o qual *p* atualmente aponta. Estas e outras construções similares são as formas mais simples e comuns da aritmética com apontadores ou endereços.

C é consistente e regular no que tange à aritmética com endereços; a integração de apontadores, arranjos e aritmética com endereços é uma das maiores vantagens da linguagem. Vamos ilustrar algumas das propriedades escrevendo um alocador de memória elementar (mas útil, apesar de sua simplicidade). Existem duas rotinas: *aloca* (*n*) que retorna um apontador *p* para *n* posições consecutivas de caracteres, as quais podem ser usadas pelo chamador de *aloca* para armazenar caracteres; *libera* (*p*) libera a área adquirida para que possa ser reutilizada. As rotinas são “elementares” porque as chamadas a *libera* devem ser feitas na ordem inversa das chamadas a *aloca*. Isto é, a área de armazenamento gerenciada por *aloca* e *libera* é uma pilha. A biblioteca padrão de C fornece funções análogas que não têm tais restrições, e, no Capítulo 8, mostraremos versões melhoradas. Até lá, entretanto, muitas aplicações necessitam apenas de um *aloca* trivial para obter pequenas áreas de armazenamento de tamanhos e em tempos imprevisíveis.

A implementação mais simples é fazer com que *aloca* forneça pedaços de um grande arranjo de caracteres que nós chamaremos *bufaloc*. Este arranjo é privado a *aloca* e *libera*. Desde que elas manipulam apontadores, e não índices de arranjos, nenhuma outra rotina precisa conhecer o nome do arranjo, o qual pode ser declarado como sendo do tipo externo estático, isto é, local ao arquivo-fonte contendo *aloca* e *libera*, e invisível fora dele. Em implementações práticas, o arranjo pode até não ter um nome; ele poderia ser obtido através de um pedido ao sistema operacional que devolva um apontador para um bloco de armazenamento sem nome.

A outra informação necessária é saber quanto de *bufaloc* já foi usado. Usamos um apontador para o próximo elemento livre, chamado *aaloc*. Quando se pede *n* caracteres a *aloca*, ela verifica se há espaço em *bufaloc*. Caso haja, *aloca* retorna o valor corrente de

aaloc (o início do bloco livre), e incrementa-o de *n* para apontar para o próximo bloco livre. libera (*p*) simplesmente atribui *p* a aaloc, se *p* é um endereço dentro de bufaloc.

```
#define NULL 0 /* valor de um apontador para fins de erro */
#define TAMALOC 1000 /* tamanho do espaco disponivel */

static char bufaloc [TAMALOC]; /* espaco para aloca */
static char *aaloc = bufaloc; /* proxima posicao livre */

char *aloca (n) /* retorna um apontador para n caracteres */
int n;
{
    if (aaloc + n <= bufaloc + TAMALOC) { /* tem espaco */
        aaloc += n;
        return (aaloc - n); /* apontador antigo */
    } else /* nao tem espaco */
        return (NULL);
}

libera (p) /* libera espaco apontado por p */
char *p;
{
    if (p >= bufaloc && p < bufaloc + TAMALOC)
        aaloc = p;
}
```

Algumas explicações seguem. Em geral, um apontador pode ser inicializado com outra variável qualquer, embora normalmente os únicos valores significativos sejam NULL (discutido abaixo) ou uma expressão envolvendo endereços de dados previamente definidos e de tipos apropriados. A declaração

```
static char *aaloc = bufaloc;
```

define aaloc como um apontador de caractere e inicializa-o para apontar para bufaloc, que é a próxima posição livre quando o programa começa. Isso poderia ser escrito como segue

```
static char *aaloc = &bufaloc [0];
```

visto que o nome do arranjo é o endereço do zero-ésimo elemento; use o que achar mais natural.

O teste

```
if (aaloc + n <= bufaloc + TAMALOC)
```

verifica se há espaço suficiente para satisfazer um pedido de *n* caracteres. Se houver, o novo valor de aaloc será no máximo um após o fim de bufaloc. Se o pedido pode ser satisfeito, aloca retorna um apontador normal (observe a declaração da função). Se não, aloca deve retornar algum sinal para indicar que não há espaço. C garante que nenhum apontador apontando para um objeto válido terá valor zero, de forma que o valor zero pode ser retornado para sinalizar um evento anormal, neste caso, a falta de espaço. Escrevemos NULL ao invés de zero, entretanto, para indicar mais claramente que este é um valor especial para um apontador. Em geral, inteiros não podem ser atribuídos a apontadores; zero é um caso especial.

Testes tais como:

```
if (aalloc + n <= bufaloc + TAMALOC)
```

e

```
if (p >= bufaloc && p < bufaloc + TAMALOC)
```

mostram várias facetas importantes da aritmética com apontadores. Primeiro, apontadores podem ser comparados sob certas circunstâncias. Se *p* e *q* apontam para membros do mesmo arranjo, então relações tais como *<*, *>*, *=*, etc. funcionam corretamente. Por exemplo,

```
p < q
```

é verdadeiro se *p* aponta para um elemento do arranjo anterior ao apontado por *q*. As relações *==* e *!=* também funcionam. Qualquer apontador pode ser comparado para testar igualdade ou desigualdade com o valor *NULL*. Entretanto, não se pode fazer aritmética ou comparações com apontadores que apontam para arranjos diferentes. Se você tiver sorte, obterá erros óbvios em todas as máquinas. Se não tiver sorte, seu código funcionará em uma máquina mas falhará misteriosamente em outra.

Segundo, já observamos que um apontador e um inteiro podem ser adicionados ou subtraídos. A construção

```
p + n
```

significa o *n*-ésimo elemento a partir da posição apontada por *p*. Isto é verdade independentemente do tipo de objeto para o qual *p* aponta; o compilador ajusta *n* de acordo com o tamanho dos objetos apontados por *p*; este tamanho é determinado pela declaração de *p*. Por exemplo, no PDP-11, os fatores de escala são 1 para *char*, 2 para *int* e *short*, 4 para *long* e *float*, e 8 para *double*.

A subtração de apontadores também é válida: se *p* e *q* apontam para membros de um mesmo arranjo, *p - q* é o número de elementos entre *p* e *q*. Este fato pode ser usado para escrever outra versão de *strlen*:

```
strlen (s) /* retorna o tamanho da cadeia s */
char *s;
{
    char *ap = s;

    while (*ap != '\0')
        ap ++;
    return (ap - s);
}
```

Nesta declaração, *ap* é inicializado com o valor de *s*, isto é, *ap* aponta para o primeiro caractere. No laço *while*, cada caractere é examinado até que o *\0* seja visto no final da cadeia. Como *\0* é zero, e como o *while* testa somente se a expressão é zero, é possível omitir o teste explícito, e escrever tais laços da seguinte forma:

```
while (*ap)
    ap ++;
```

Desde que *ap* aponta para caracteres, *ap ++* avança *ap* para o próximo caractere de cada vez, e *ap - s* dá o número de caracteres pulados, isto é, o tamanho da cadeia. A aritmética com apontadores é consistente: se estivermos manipulando objetos do tipo

float, que ocupam mais espaço que char, e se ap fosse um apontador para float, ap++ avançaria para o próximo float. Poderíamos então escrever uma outra versão de aloca para manipular, digamos, float ao invés de char, simplesmente mudando char para float em aloca e libera. Toda manipulação de apontadores levaria em conta automaticamente o tamanho do objeto apontado, de forma que nada mais teria de ser mudado.

Quaisquer operações além das mencionadas aqui (adição ou subtração de apontador com um inteiro; subtração ou comparação de dois apontadores), são ilegais. Não é permitido adicionar dois apontadores, ou multiplicar ou dividir ou deslocar ou mascarar, ou adicionar float ou double aos mesmos.

5.5 Apontadores de Caractere e Funções

Uma *constante do tipo cadeia*, escrita da seguinte forma:

`"Eu sou uma cadeia"`

é um arranjo de caracteres. Na representação interna, o compilador termina o arranjo com o caractere `\0` para que programas possam encontrar o fim. O tamanho de armazenamento é portanto um a mais do que o número de caracteres entre aspas.

Talvez a ocorrência mais comum de constantes do tipo cadeia seja como argumentos para funções, como em:

```
printf("primeiro programa\n");
```

Quando uma cadeia de caracteres como esta aparece num programa, o acesso à mesma é feito através de um apontador de caractere; `printf` recebe um apontador para o arranjo de caracteres.

Arranjos de caracteres não são sempre argumentos de uma função. Se mensagem é declarada como segue

```
char *mensagem;
```

então o comando

```
mensagem = "O sertanejo é, antes de tudo, um forte.";
```

atribui a mensagem um apontador para os caracteres da cadeia. Isto *não* é uma cópia da cadeia; somente apontadores estão envolvidos. C não fornece qualquer operador para processar uma cadeia de caracteres como um todo.

Ilustraremos outros aspectos de apontadores e arranjos pelo estudo de duas funções úteis da biblioteca padrão de entrada e saída a ser discutida no Capítulo 7.

A primeira função é `strcpy(s, t)` que copia a cadeia `t` na cadeia `s`. Os argumentos são escritos nesta ordem por analogia a uma atribuição, onde diríamos:

```
s = t
```

para atribuir `t` a `s`. A versão com arranjos é:

```
strcpy(s, t) /* copia t a s */
char s[ ], t[ ];
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Compare esta versão com a seguinte que usa apontadores

```
strcpy (s, t) /* copia t a s; versao 1 com apontadores */
char *s, *t;
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Como os argumentos são passados por valor, strcpy pode usar s e t como quiser. Aqui eles são apontadores convenientemente inicializados, que percorrem os arranjos um caractere por vez, até que o \0 que termina t tenha sido copiado em s.

Na prática, strcpy não seria escrita como mostrada acima. Uma segunda possibilidade seria

```
strcpy (s, t) /* copia t a s; versao 2 com apontadores */
char *s, *t;
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Esta versão move o incremento de s e t para a parte de teste do laço. O valor de *t++ é o caractere para o qual t aponta antes do incremento de t; o ++ pós-fixado não muda t até que o caractere tenha sido apanhado. Da mesma forma, o caractere é armazenado na posição para a qual s aponta antes do mesmo ser incrementado. Esse caractere é também o valor comparado com \0 para controlar o laço. O resultado é a cópia dos caracteres de t para s até o \0 final, inclusive.

Como uma abreviação final, observemos que a comparação com \0 é redundante, e a função é frequentemente escrita como segue

```
strcpy (s, t) /* copia t a s; versao 3 com apontadores */
char *s, *t;
{
    while (*s++ = *t++)
        ;
}
```

Embora essa forma pareça complicada à primeira vista, a conveniência de notação é considerável, e o idioma deve ser dominado, inclusive porque você verá esta construção frequentemente em programas C.

A segunda rotina é strcmp (s, t), que compara as cadeias de caracteres s e t, e retorna um valor negativo, zero ou positivo se s é lexicograficamente menor, igual ou maior que t, respectivamente. O valor retornado é obtido pela subtração dos caracteres na primeira posição onde s e t são diferentes.

```
strcmp (s, t) /* retorna < 0 se s < t, 0 se s == t, > 0 se s > t */
char s [ ], t [ ];
{
    int i;
```

```

i = 0;
while (s[i] == t[i])
    if (s[i++] == '\0')
        return (0);
return (s[i] - t[i]);
}

```

A versão com apontadores é:

```

strcmp (s, t) /* retorna < 0 se s < t, 0 se s == t, > 0 se s > t */
char *s, *t;
{
    for (; *s == *t; s++, t++)
        if (*s == '\0')
            return (0);
    return (*s - *t);
}

```

Desde que ++ e -- são ambos operadores pré ou pós-fixados, outras combinações de * com ++ e -- ocorrem, embora com menos frequência. Por exemplo:

* ++ p

incrementa p antes de obter o caractere para o qual p aponta;

* -- p

decrementa p primeiro.

Exercício 5-2. Escreva uma versão com apontadores da função `strcat` que mostramos no Capítulo 2: `strcat (s, t)` copia a cadeia t no fim da cadeia s.

Exercício 5-3. Escreva uma macro para `strcpy`.

Exercício 5-4. Reescreva programas apropriados dos capítulos e exercícios anteriores com apontadores ao invés de indexação de arranjos. Boas possibilidades incluem `telinha` (Capítulos 1 e 4), `atoi`, `itoa`, e suas variantes (Capítulos 2, 3, e 4), `inverte` (Capítulo 3), e `index` e `obtemop` (Capítulo 4).

5.6 Apontadores Não São Inteiros

Você pode observar em programas C antigos, uma atitude cavalheiresca na cópia de apontadores. Na maioria das máquinas, um apontador pode ser atribuído a um inteiro e vice-versa, sem alterá-lo; não tem mudança de escala, ou conversão, e não tem perda de bits. Lamentavelmente, isso tem levado a certas liberdades em rotinas que retornam apontadores que são então passados para outras rotinas — as declarações necessárias de apontadores são freqüentemente omitidas. Por exemplo, considere a função `guardacad (s)`, que guarda a cadeia s num lugar, obtido por uma chamada a `aloca`, e retorna um apontador para o mesmo. De forma correta, ela seria escrita como segue

```

char *guardacad (s) /* guarda a cadeia s em algum lugar */
char *s;
{
    char *ap, *aloca ();

```

```

    if ((ap = aloca (strlen (s) + 1)) != NULL)
        strcpy (ap, s);
    return (ap);
}

```

Na prática, há uma forte tendência de omitir declarações:

```

guardacad (s) /* guarda a cadeia s em algum lugar */
{
    char *ap;

    if ((ap = aloca (strlen (s) + 1)) != NULL)
        strcpy (ap, s);
    return (ap);
}

```

Isso funcionará em muitas máquinas, visto que o tipo assumido para funções e argumentos é int, e int e apontador podem ser atribuídos um ao outro livremente. No entanto esse tipo de código é perigoso, pela sua dependência dos detalhes de implementação e arquitetura de máquina, os quais podem não se aplicar ao compilador particular sendo usado. É melhor ser completo em todas as declarações. (O programa *lint* avisará da ocorrência de tais construções, quando elas aparecerem inadvertidamente.)

5.7 Arranjos Multidimensionais

C provê arranjos multidimensionais, embora tendam a ser muito menos usados na prática que arranjos de apontadores. Nesta seção, veremos algumas de suas propriedades.

Considere o problema de conversão de data, de dia do mês para dia do ano e vice-versa. Por exemplo, o primeiro de março é o sexagésimo dia de um ano normal e o sexagésimo primeiro de um ano bissexto. Vamos definir duas funções para fazer as conversões: *dia-do-ano* converte mês e dia em dia do ano, e *dia-mes* converte o dia do ano em mês e dia. Desde que essa última função retorne dois valores, os argumentos mês e dia serão apontadores:

```

dia_mes (1977, 60, &m, &d)

```

atribui 3 a m e 1 a d (primeiro de março).

Ambas as funções necessitam da mesma informação, uma tabela do número de dias em cada mês. Desde que o número de dias por mês difere para anos normais e bissextos, é mais fácil separá-los em duas linhas de um arranjo bidimensional do que tentar ajustar a contagem para fevereiro durante o cálculo. O arranjo e as funções que fazem as conversões são as seguintes:

```

static int tab_dia [2] [13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

dia_do_ano (ano, mes, dia) /* acha o dia do ano */
int ano, mes, dia; /* a partir do mes e dia do mes */
{
    int i, bissexto;

```

```

    bissexto = ano%4 == 0 && ano%100 != 0 || ano%400 == 0;
    for (i = 1; i < mes; i++)
        dia += tab_dia[bissexto][i];
    return (dia);
}

dia_mes(ano, dia_do_ano, ames, adia) /* acha mes e dia */
int ano, dia_do_ano, *ames, *adia; /* a partir do dia do ano */
{
    int i, bissexto;

    bissexto = ano%4 == 0 && ano%100 != 0 || ano%400 == 0;
    for (i = 1; dia_do_ano > tab_dia[bissexto][i]; i++)
        dia_do_ano -= tab_dia[bissexto][i];
    *ames = i;
    *adia = dia_do_ano;
}

```

O arranjo `tab_dia` deve ser externo para `dia_do_ano` e `dia_mes` para que ambos possam usá-lo.

`tab_dia` é o primeiro arranjo bidimensional que nós usamos. Em C, por definição, um arranjo bidimensional é na realidade um arranjo unidimensional, onde cada elemento é um arranjo. Por isso índices são escritos como:

```
tab_dia[i][j]
```

ao invés de

```
tab_dia[i, j]
```

como na maioria das linguagens. Fora isso, um arranjo bidimensional pode ser tratado da mesma forma que em outras linguagens. Elementos são armazenados por linha, isto é, o índice mais à direita varia mais rapidamente quando os elementos são acessados na ordem de armazenamento.

Um arranjo é inicializado por uma lista de valores iniciais entre chaves; cada linha de um arranjo bidimensional é inicializada por uma sublista correspondente. Nós iniciamos o arranjo `tab_dia` com uma coluna de zeros para que os números dos meses possam variar de forma natural de 1 a 12 e não de 0 a 11. Visto que o espaço de memória não é uma preocupação aqui, isso é mais fácil do que ajustar índices.

Se um arranjo bidimensional *deve* ser passado para uma função, a declaração do argumento na função deve incluir a dimensão da coluna; a dimensão da linha é irrelevante, desde que o que é passado seja um apontador. Neste caso particular, é um apontador para objetos que são arranjos de 13 elementos inteiros. Então, se o arranjo `tab_dia` for passado para a função `f`, a declaração de `f` seria:

```

f(tab_dia)
int tab_dia[2][13];
{
    ...
}

```

A declaração do argumento em `f` poderia ser

```
int tab_dia[][13];
```


desde que o número de linhas é irrelevante, ou poderia ser

```
int (*tab_dia) [13];
```

o qual diz que o argumento é um apontador para um arranjo de 13 inteiros. Os parênteses são necessários porque colchetes têm maior precedência que *; sem parênteses, a declaração

```
int *tab_dia [13];
```

é um arranjo de 13 apontadores para inteiros, como veremos na próxima seção.

5.8 Arranjos de Apontadores; Apontadores para Apontadores

Visto que apontadores são variáveis, pode-se esperar o uso de arranjos de apontadores. Isto, de fato ocorre. Vamos ilustrar tais construções com um programa que ordene um conjunto de linhas de texto em ordem alfabética, uma versão reduzida do utilitário *sort* do UNIX.

No Capítulo 3 apresentamos uma função de ordenação Shell que ordena um arranjo de inteiros. O mesmo algoritmo funcionará, exceto que agora temos de manipular linhas de texto de tamanhos diferentes, e que, ao contrário de inteiros, não podem ser comparadas ou movidas com uma única operação. Precisamos de uma representação de dados que seja eficiente e conveniente para linhas de tamanho variável.

É neste ponto que o arranjo de apontadores é útil. Se as linhas a serem ordenadas estiverem armazenadas contiguamente num longo arranjo de caracteres (mantido por alocação, talvez), então cada linha pode ser acessada através de um apontador para seu primeiro caractere. Os apontadores podem ser armazenados num arranjo. Duas linhas podem ser comparadas passando-se seus apontadores para *strcmp*. Quando duas linhas fora de ordem tiverem de ser trocadas, os *apontadores* no arranjo de apontadores são mudados, e não as próprias linhas do texto. Isso elimina o duplo problema de um gerenciamento de espaço complicado e uma alta sobrecarga de trabalho resultante da movimentação de linhas.

O processo de ordenação envolve três passos:

ler todas as linhas de entrada

ordena-las

imprimi-las em ordem

Como sempre, é melhor dividir o programa em funções que reflitam esta divisão natural, com a rotina principal controlando o processo.

Vamos deixar de lado o passo de ordenação por um momento, e nos concentrar na estrutura de dados e na entrada e saída. A rotina de entrada tem de coletar e armazenar os caracteres de cada linha, e montar um arranjo de apontadores para essas linhas. Ela terá também de contar o número de linhas de entrada, visto que essa informação seja necessária para a ordenação e impressão. Como a função de entrada dá conta apenas de um número finito de linhas de entrada, ela pode retornar algum valor ilegal de contagem de linhas, tal como -1, no caso de excesso de linhas de entrada. A rotina de saída tem somente de imprimir as linhas na ordem em que elas aparecem no arranjo de apontadores.

```
#define NULL 0
```

```
#define LINHAS 100 /* numero maximo de linhas a ordenar */
```

```

main ( ) /* ordena as linhas de entrada */
{
    char *alinha [LINHAS]; /* apontadores para as linhas */
    int nlinhas; /* numero de linhas lidas */

    if ((nlinhas = obtlinhas (alinha, LINHAS)) >= 0)
        ordena (alinha, nlinhas);
        imprlinhas (alinha, nlinhas);
    }
    else
        printf ("linhas em excesso na entrada\n");
}

```

```

#define TAMMAX 1000

```

```

obtlinhas (alinha, maxlinhas) /* le linhas de entrada */
char *alinha [ ]; /* para ordenar */
int maxlinhas;
{
    int tam, nlinhas;
    char *ap, *aloca ( ), linha [TAMMAX];

    nlinhas = 0;
    while ((tam = lelinha (linha, TAMMAX)) > 0)
        if (nlinhas >= maxlinhas)
            return (-1);
        else if ((ap = aloca (tam)) == NULL)
            return (-1);
        else {
            linha[tam-1] = '\0'; /* remove nova-linha */
            strcpy (ap, linha);
            alinha [nlinhas + +] = ap;
        }
    return (nlinhas);
}

```

O caractere de nova-linha no final de cada linha é removido para que não afete a ordem em que elas são ordenadas.

```

imprlinha (alinha, nlinhas) /* imprime as linhas de saída */
char *alinha [ ];
int nlinhas;
{
    int i;
    for (i = 0; i < nlinhas; i++)
        printf ("%s\n", alinha[i]);
}

```

A principal novidade é a declaração de *alinha*:

```
char *alinha [LINHAS];
```

que diz que *alinha* é um arranjo de LINHAS elementos, cada um dos quais é um apontador para char. Isto é, *alinha* [i] é um apontador de caractere, e **alinha* [i] acessa um caractere.

Desde que *alinha* é um arranjo que é passado para *imprlinhas*, ele pode ser tratado como um apontador da mesma forma que em nossos exemplos anteriores, e a função pode ser escrita como:

```
imprlinhas (alinha, nlinhas) /* imprime as linhas de saída */
char *alinha [ ];
int nlinhas;
{
    while (-- nlinhas >= 0)
        printf ("%s\n", *alinha + +);
}
```

**alinha* aponta inicialmente para a primeira linha; cada incremento avança-o para a próxima linha enquanto *nlinhas* é decrementado.

Com a entrada e saída sob controle, podemos prosseguir com a ordenação. O algoritmo Shell do Capítulo 3 precisa de poucas mudanças: as declarações devem ser modificadas, e a operação de comparação deve ser feita por uma função separada. O algoritmo básico permanece o mesmo, o que nos dá alguma confiança que funcionará corretamente.

```
ordena (v, n) /* ordena as cadeias v [0] . . . v [n - 1] */
char *v [ ]; /* em ordem crescente */
int n;
{
    int inter, i, j;
    char *temp;

    for (inter = n/2; inter > 0; inter /= 2)
        for (i = inter; i < n; i + +)
            for (j = i - inter; j >= 0; j -= inter) {
                if (strcmp (v [j], v [j + inter]) <= 0)
                    break;
                temp = v [j];
                v [j] = v [j + inter];
                v [j + inter] = temp;
            }
}
```

Como cada elemento de *v* (*alinha*) é um apontador de caractere, *temp* também o deve ser, de forma que um possa ser copiado no outro.

Nós escrevemos o programa tão diretamente quanto possível, de modo a fazê-lo funcionar rapidamente. Ele poderia ser mais rápido, por exemplo, copiando as linhas de entrada diretamente num arranjo mantido por *obtlinhas*, ao invés de copiá-las em linha e então num local escondido, mantido por *aloca*. Mas é melhor tomar o primeiro projeto fácil de entender, e se preocupar com "eficiência" mais tarde. A forma de aumentar significativamente a rapidez deste programa provavelmente não é evitando uma cópia desne-

cessária das linhas de entrada, mas trocando o algoritmo de ordenação Shell por algo melhor tal como o Quicksort.

No Capítulo 1, mostramos que visto que os laços `while` e `for` testam a condição de término antes de executar o corpo do laço, eles ajudam a garantir que programas funcionem nos casos limites, em particular quando nenhuma entrada é fornecida. Vale a pena examinar as funções do programa de ordenação, verificando-se o que acontece quando não há texto de entrada.

Exercício 5-5. Reescreva `obtlinhas` para criar linhas num arranjo fornecido por `main`, ao invés de chamar `aloca`. Quão mais rápido é o programa?

5.9 Inicialização de Arranjos de Apontadores

Considere o problema de escrever uma função `nome-do-mes (n)` que retorna um apontador para uma cadeia de caracteres contendo o nome do n -ésimo mês. Esta é uma aplicação ideal para um arranjo estático interno. `nome_do_mes` contém um arranjo privado de cadeias de caracteres, e retorna um apontador para o nome apropriado quando chamada. A seção presente aborda o tema de inicialização deste arranjo de nomes.

A sintaxe é semelhante às inicializações anteriores:

```
char *nome_do_mes (n) /* retorna nome do n-esimo mes */
int n;
{
    static char *nome [ ] = {
        "mes ilegal",
        "janeiro",
        "fevereiro",
        "marco",
        "abril",
        "maio",
        "junho",
        "julho",
        "agosto",
        "setembro",
        "outubro",
        "novembro",
        "dezembro"
    };
    return ((n < 1 || n > 12) ? nome [0] : nome [n]);
}
```

A declaração de `nome`, o qual é um arranjo de apontadores de caractere, é idêntica à de `alinha` no exemplo de ordenação. O inicializador é simplesmente uma lista de cadeias de caracteres; cada uma é associada à posição correspondente do arranjo. Mais precisamente, os caracteres da i -ésima cadeia são colocados em algum lugar e um apontador para eles é armazenado em `nome [i]`. Como o tamanho do arranjo `nome` não é especificado, o próprio compilador conta os inicializadores e usa este valor como o tamanho do arranjo.

5.10 Apontadores Versus Arranjos Multidimensionais

Novos usuários de C ficam confusos às vezes sobre a diferença entre um arranjo bi-

dimensional e um arranjo de apontadores, tal como nome no exemplo acima. Dadas as declarações

```
int a [10][10];  
int *b [10];
```

o uso de *a* e *b* pode ser semelhante, já que *a* [5] [5] e *b* [5] [5] são referências legais a um int. Mas *a* é verdadeiramente um arranjo: 100 células de armazenamento foram alocadas, e o cálculo convencional retangular de índices é feito para se encontrar um dado elemento. Para *b*, entretanto, a declaração aloca somente 10 apontadores; cada um deve ser inicializado para apontar para um arranjo de inteiros. Presumindo-se que cada um aponte para um arranjo de dez elementos, haverá então 100 células de armazenamento alocadas, mais dez células para os apontadores. Assim, o arranjo de apontadores usa um pouco mais de espaço, e pode necessitar de uma inicialização explícita. Mas ele tem duas vantagens: o acesso a um elemento é feito indiretamente através de um apontador ao invés de usar uma multiplicação e uma adição, e as linhas do arranjo podem ter tamanhos diferentes. Isto é, cada elemento de *b* não precisa apontar para um vetor de dez elementos; algum pode apontar para um vetor de dois elementos, outro para um de vinte, outros, ainda, para nada.

Embora tenhamos discutido em termos de inteiros, o uso mais freqüente de arranjos de apontadores é o mostrado em nome-do-mes para armazenar cadeias de caracteres de diversos tamanhos.

Exercício 5-6. Reescreva as rotinas *dia-do-ano* e *dia-mes* com apontadores ao invés de indexação.

5.11 Argumentos da Linha de Comando

Nos ambientes que suportam C, há um modo de se passar argumentos da linha de comando ou parâmetros para um programa quando o mesmo começa sua execução. Quando *main* é chamada para iniciar a execução, ela é chamada com dois argumentos. O primeiro (convencionalmente chamado *argc*) é o número de argumentos com que o programa foi chamado; o segundo (*argv*) é um apontador para um arranjo de cadeias de caracteres que contêm os argumentos, um por cadeia. A manipulação dessas cadeias de caracteres é um uso comum de múltiplos níveis de apontadores.

A ilustração mais simples das declarações necessárias e do uso de tais variáveis, é o programa *echo*, que simplesmente ecoa seus argumentos da linha de comando numa única linha, separados por brancos. Isto é, se o comando

```
echo primeiro programa
```

é dado, a saída é

```
primeiro programa
```

Por convenção, *argv* [0] é o nome pelo qual o programa foi ativado, de forma que *argc* é pelo menos 1. No exemplo acima *argc* é 3, e *argv* [0], *argv* [1] e *argv* [2] são “echo”, “primeiro”, e “programa”, respectivamente. O primeiro argumento real é *argv* [1] e o último é *argv* [*argc* - 1]. Se *argc* é 1, não há argumentos após o nome do programa. Isso é mostrado em *echo*:

```
main (argc, argv) /* eco dos argumentos; versao 1 */  
int argc;
```

```
char *argv [ ];
{
    int i;

    for (i = 1; i < argc; i++)
        printf ("%s%c", argv[i], (i < argc - 1) ? ' ' : '\n');
}
```

Como `argv` é um apontador para um arranjo de apontadores, há muitas maneiras de se escrever este programa com a manipulação de apontadores ao invés da indexação de um arranjo. Vamos mostrar duas variações:

```
main (argc, argv) /* eco dos argumentos; versao 2 */
int argc;
char *argv [ ];
{
    while (--argc > 0)
        printf ("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

Como `argv` é um apontador para o início do arranjo de cadeias de argumento, incrementá-lo de 1 (`++argv`) faz com que ele aponte para o `argv[1]` original ao invés de `argv[0]`. Cada incremento sucessivo o move para o próximo argumento; `*argv` é então o apontador para o argumento. Ao mesmo tempo, `argc` é decrementado; quando torna-se zero, não há mais argumentos para imprimir.

Alternativamente, podemos escrever

```
main (argc, argv) /* eco dos argumentos; versao 3 */
int argc;
char *argv [ ];
{
    while (--argc > 0)
        printf ((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

Esta versão mostra que o argumento de formato de `printf` pode ser uma expressão como qualquer outra. Este uso não é muito freqüente, mas deve ser lembrado.

Como segundo exemplo, vamos fazer algumas mudanças no programa de pesquisa de padrão do Capítulo 4. Se você se lembrar, nós embutimos o padrão de pesquisa dentro do programa, uma solução obviamente não satisfatória. Apoiando-se na sintaxe do utilitário *grep* do UNIX, vamos mudar o programa para que o padrão a ser pesquisado seja especificado pelo primeiro argumento da linha de comando.

```
#define MAXLINHA 1000

main (argc, argv) /* acha padrao a partir do argumento 1 */
int argc;
char *argv [ ];
{
    char linha [MAXLINHA];
```

```

if (argc != 2)
    printf ("Sintaxe: acha padrao\n");
else
    while (lelinha (linha, MAXLINHA) > 0)
        if (index (linha, argv [1]) >= 0)
            printf ("%s", linha);
}

```

O modelo básico pode ser elaborado agora para ilustrar outras construções com apontadores. Suponha que nós desejamos permitir dois argumentos opcionais. Um diz "imprima todas as linhas *exceto* aquelas que contenham o padrão"; o segundo diz "preceda cada linha impressa com seu número".

Uma convenção comum para programas C, é a de que um argumento começando com um sinal de menos introduz um sinalizador ou parâmetro opcionais. Se nós escolhermos - x ("exceto") para sinalizar a inversão, e - n ("numerar") para requisitar a numeração das linhas, então o comando

acha - x - n an

com a entrada

O sertanejo é, antes de tudo, um forte.
 A sua aparência, entretanto,
 ao primeiro lance de vista,
 revela o contrário.

deve produzir como saída

4: revela o contrário.

Argumentos opcionais devem ser permitidos em qualquer ordem, e o resto do programa deve ser insensível ao número de argumentos que estão presentes. Em particular, a chamada para index não deve se referir a argv [2] quando houver um único parâmetro opcional e a argv [1] quando não houver. Além do mais, é conveniente para os usuários que argumentos opcionais possam ser concatenados como em:

acha - nx an

O programa segue.

```
#define MAXLINHA 1000
```

```
main (argc, argv) / * acha padrao a partir do primeiro argumento */
```

```
int argc;
```

```
char *argv [ ];
```

```
{
```

```
    char linha [MAXLINHA], *s;
```

```
    long num_linha = 0;
```

```
    int excluir = 0, numerar = 0;
```

```
    while (--argc > 0 && (*++argv) [0] != '-')
```

```
        for (s = argv [0] + 1; *s != '\0'; s++)
```

```
            switch (*s) {
```

```
                case 'x':
```

```

        excluir = 1;
        break;
    case 'n':
        numerar = 1;
        break;
    default:
        printf ("acha: opcao %c ilegal\n", *s);
        argc = 0;
        break;
    }
if (argc != 1)
    printf ("Sintaxe: acha - x - n padrão\n");
else
    while (linha (linha, MAXLINHA) > 0) {
        num_linha ++;
        if ((index (linha, *argv) >= 0) != excluir) {
            if (numerar)
                printf ("%ld: ", num_linha);
            printf ("%s", linha);
        }
    }
}

```

`argv` é incrementado antes de cada argumento opcional, e `argc` é decrementado. Se não há erros, no final do laço, `argc` deve ser 1 e `*argv` deve apontar para o padrão. Observe que `*++argv` é um apontador para uma cadeia argumento; `(*++argv)[0]` é seu primeiro caractere. Os parênteses são necessários, pois sem eles a expressão seria `*+ + (argv[0])` o que é bastante diferente (e errado). Uma forma alternativa válida seria `**+ + argv`.

Exercício 5-7. Escreva o programa `soma` que avalia uma expressão polonesa reversa dada na linha de comando. Por exemplo:

`soma 2 3 4 + *`

avalia $2 \times (3 + 4)$.

Exercício 5-8. Modifique os programas `tab` e `destab` (escritos como exercícios no Capítulo 1) para aceitarem uma lista de pontos de tabulação como argumentos. Use os pontos normais de tabulação, se não forem dados argumentos.

Exercício 5-9. Estenda `tab` e `destab` para aceitarem a abreviação

`tab m +n`

que define pontos de tabulação a cada n colunas a partir da coluna m . Escolha um comportamento conveniente (para o usuário) caso não sejam dados argumentos.

Exercício 5-10. Escreva o programa `cauda`, que imprima as últimas n linhas de sua entrada. Caso não especificado, n assume o valor 10, digamos, mas ele pode ser alterado por um argumento opcional, de forma que

`cauda - n`

imprima as últimas n linhas. O programa deve se comportar de forma racional independente da entrada e do valor de n . Escreva o programa de modo que ele faça o melhor uso da área de armazenamento disponível: linhas devem ser armazenadas como na função *ordena*, sem usar um arranjo bidimensional de tamanho fixo.

5.12 Apontadores para Funções

Em C, uma função não é uma variável, mas é possível se definir um *apontador para uma função*, o qual pode ser manipulado, passado como argumento, colocado em arranjos, e assim por diante. Ilustraremos isso modificando o programa de ordenação escrito anteriormente neste capítulo para que, na presença de um argumento opcional — n , ele ordene as linhas de entrada numericamente e não lexicograficamente.

Um ordenador frequentemente consiste de três partes — uma *comparação* que determina a ordem de qualquer par de objetos, uma *troca* que inverte sua ordem, e um *algoritmo de ordenação* que faz as comparações e trocas até que os objetos estejam ordenados. O algoritmo de ordenação é independente das operações de comparação e de troca de modo que, passando diferentes funções de comparação e de troca para ele, podemos ordenar de acordo com diferentes critérios. Este é o enfoque tomado no novo ordenador.

A comparação lexicográfica de duas linhas é feita por *strcmp* e a troca por troca como vimos anteriormente; precisaremos também de uma rotina *compnum* que compare duas linhas numericamente e retorne o mesmo tipo de condição que *strcmp* retorna. Essas três funções são declaradas em *main* e apontadores para elas são passados para o ordenador *ordena* que as chama via apontadores. Não incluímos o processamento de erros para os argumentos, para nos concentrarmos nas idéias principais.

```
#define LINHAS 100 /* numero maximo de linhas a ordenar */
```

```
main (argc, argv) /* ordena as linhas de entrada */
int argc;
char *argv [ ];
{
    char *alinha [LINHAS]; /* apontadores para as linhas */
    int nlinhas; /* numero de linhas lidas */
    int strcmp ( ), compnum ( ); /* funcoes de comparacao */
    int troca ( ); /* funcao de permuta */
    int numerica = 0; /* 1 se ordenacao numerica */

    if (argc > 1 && argv [1] [0] == '-' && argv [1] [1] == 'n')
        numerica = 1;
    if ((nlinhas = obtlinhas (alinha, LINHAS)) >= 0) {
        if (numerica)
            ordena (alinha, nlinhas, compnum, troca);
        else
            ordena (alinha, nlinhas, strcmp, troca);
        imprlinhas (alinha, nlinhas);
    } else
        printf ("linhas em excesso na entrada\n");
}
```

strcmp, compnum e troca são endereços de funções; visto que elas são conhecidas como sendo funções, o operador & não é necessário, da mesma forma que ele não é necessário antes do nome de um arranjo. O compilador faz com que o endereço da função seja passado.

O segundo passo é a modificação de ordena:

```
ordena (v, n, comp, perm) /* ordena as cadeias v [0] ... v [n - 1] */
char *v [ ]; /* em ordem crescente */
int n;
int (*comp) ( ), (*perm) ( );
{
    int inter, i, j;

    for (inter = n/2; inter > 0; inter /= 2)
        for (i = inter; i < n; i++)
            for (j = i - inter; j >= 0; j -= inter) {
                if ((*comp) (v [j], v [j + inter]) <= 0)
                    break;
                (*perm) (&v [j], &v [j + inter]);
            }
}
```

As declarações devem ser estudadas com algum cuidado

```
int (*comp) ( );
```

diz que comp é um apontador para uma função que retorna um int. O primeiro par de parênteses é necessário; sem eles,

```
int *comp ( );
```

diria que comp é uma função que retorna um apontador para um int, o que é muito diferente.

O uso de comp na linha

```
if ((*comp) (v [j], v [j + inter]) <= 0)
```

é consistente com a declaração: comp é um apontador para uma função, *comp é a função, e

```
(*comp) (v [i], v [i + inter])
```

é a chamada para a mesma. Os parênteses são necessários para que os componentes sejam corretamente associados.

Nós já mostramos strcmp, que compara duas cadeias. Vejamos compnum, que compara duas cadeias baseando-se nos valores numéricos iniciais:

```
compnum (s1, s2) /* compara s1 e s2 numericamente */
char *s1, *s2;
{
    double atof ( ), v1, v2;

    v1 = atof (s1);
    v2 = atof (s2);
    if (v1 < v2)
```

```

        return (- 1);
    else if (v1 > v2)
        return (1);
    else
        return (0);
}

```

O passo final é acrescentar a função *troca* que troca dois apontadores. Ela é adaptada diretamente da versão apresentada anteriormente neste capítulo.

```

troca (ax, ay) /* permuta *ax e *ay */
char *ax [ ], *ay [ ];
{
    char *temp;

    temp = *ax;
    *ax = *ay;
    *ay = temp;
}

```

Há uma variedade de outras opções que podem ser acrescentadas ao programa de ordenação; algumas são exercícios desafiadores.

Exercício 5-11. Modifique *ordena* para manipular um sinalizador *-r* que indica uma ordenação reversa (em ordem decrescente). É claro, *-r* deve funcionar com *-n*.

Exercício 5-12. Acrescente a opção *-j* para juntar letras minúsculas e maiúsculas, de modo que elas não tenham tratamento diferenciado durante a ordenação: caracteres maiúsculos e minúsculos são ordenados juntos, de forma que *a* e *A* são adjacentes, e não separados pelo alfabeto inteiro.

Exercício 5-13. Acrescente a opção *-d* (“ordem de dicionário”), que faz com que as comparações sejam feitas somente com letras, dígitos e espaços. Assegure que *-d* funcione com *-j*.

Exercício 5-14. Acrescente uma capacidade de manipulação de campos, de forma que a ordenação possa ser feita sobre campos de uma linha, cada campo com um conjunto independente de opções. (O índice deste livro foi ordenado com *-dj* para a categoria de índice e *-n* para o número da página.)

Capítulo 6

ESTRUTURAS

Uma *estrutura* é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome para manipulação conveniente. (Estruturas são chamadas “registros” em algumas linguagens mais notadamente Pascal.)

O exemplo tradicional de uma estrutura é o registro de uma folha de pagamento: um “empregado” é descrito por um conjunto de atributos tais como nome, endereço, número do CPF, salário etc. Alguns desses, por sua vez, poderiam ser estruturas: um nome tem vários componentes, assim como um endereço e até mesmo um salário.

Estruturas auxiliam na organização de dados complexos, particularmente em grandes programas, porque em muitas situações elas permitem que um agrupamento de variáveis inter-relacionadas seja tratado como um todo ao invés de como entidades separadas. Neste capítulo, ilustraremos o uso de estruturas. Os programas que usaremos são maiores que muitos outros no livro, mas ainda de tamanho modesto.

6.1 Elementos Básicos

Vamos revisar as rotinas de conversão de datas do Capítulo 5. Uma data consiste de várias partes, tais como o dia, o mês, o ano, e talvez o dia do ano e o nome do mês. Estas cinco variáveis podem ser colocadas numa estrutura como segue:

```
struct data {  
    int dia;  
    int mes;  
    int ano;  
    int dia_ano;  
    char nome_mes[4];  
};
```

A palavra chave *struct* introduz a declaração da estrutura, que é uma lista de declarações entre chaves. Um nome opcional chamado *etiqueta de estrutura* pode seguir a palavra *struct* (como *data* aqui). A etiqueta atribui um nome a este tipo de estrutura, e pode ser usada posteriormente como uma abreviação para a descrição detalhada.

Os elementos ou variáveis mencionados na estrutura são chamados *membros*. Um membro ou etiqueta de uma estrutura e uma variável ordinária (isto é, não membro) podem ter o mesmo nome sem conflito, visto que eles podem sempre ser distinguidos pelo contexto. Evidentemente, por uma questão de estilo de programação o mesmo nome será normalmente usado apenas para objetos estreitamente relacionados.

O fecha-chave que termina a lista de membros pode ser seguida por uma lista de variáveis, exatamente como no caso de um tipo básico. Isto é, a declaração

```
struct { . . . } x, y, z;
```

é sintaticamente análoga a

```
int x, y, z;
```

no sentido de que cada comando declara *x*, *y* e *z* como sendo variáveis do tipo indicado e causa a alocação de espaço para as mesmas.

Uma declaração de estrutura que não é seguida por uma lista de variáveis não aloca espaço; ela simplesmente descreve o *gabarito* ou forma de uma estrutura. Se a declaração é etiquetada, entretanto, a etiqueta pode ser usada posteriormente em definições de instâncias da estrutura. Por exemplo, dada a declaração de data acima,

```
struct data d;
```

define a variável *d* como sendo uma estrutura do tipo *data*. Uma estrutura externa ou estática pode ser inicializada seguindo-se sua definição por uma lista de valores iniciais para os componentes:

```
struct data d = { 7, 9, 1822, 250, "set" };
```

Um membro de uma estrutura é referenciado numa expressão por uma construção da forma

nome-da-estrutura . membro

O operador de membro de estrutura "." conecta o nome da estrutura e o nome do membro. Para atribuir um valor à variável *bissexto* a partir da data na estrutura *d*, faríamos:

```
bissexto = d.ano % 4 == 0 && d.ano % 100 != 0  
          || d.ano % 400 == 0;
```

Para verificar o nome do mês, faríamos:

```
if (strcmp(d.nome_mes, "ago") == 0) . . .
```

Para converter o primeiro caractere do nome do mês para minúsculo, faríamos:

```
d.nome_mes[0] = minusculo(d.nome_mes[0]);
```

Estruturas podem estar aninhadas; um registro de folha de pagamento poderia ser:

```
struct pessoa {  
    char    nome[TAMNOME];  
    char    endereco[TAMENDER];  
    long    cep;  
    long    cpf;  
    double  salario;  
    struct  data nascimento;  
    struct  data contratacao;  
};
```

A estrutura *pessoa* contém duas datas. Se declararmos *emp* como

```
struct pessoa emp;
```

então

`emp.nascimento.mes`

refere-se ao mês de nascimento. O operador "." é associativo da esquerda para a direita.

6.2 Estruturas e Funções

Há um número de restrições sobre estruturas C. As regras essenciais são que as únicas operações que podem ser aplicadas a uma estrutura são: obter seu endereço com `&`, e acessar um dos seus membros. Isto implica que estruturas não podem ser atribuídas ou copiadas como uma unidade, e que elas não podem ser passadas para ou retornadas de funções. (Tais restrições serão removidas em versões futuras.) Apontadores para estruturas não sofrem essas limitações, entretanto, e estruturas e funções podem trabalhar juntas confortavelmente. Finalmente, estruturas automáticas, como arranjos automáticos, não podem ser inicializadas; só se pode inicializar estruturas externas ou estáticas.

Vamos investigar alguns desses pontos, reescrevendo as funções de conversão de data do último capítulo para que usem estruturas. Desde que as regras proibem a passagem de uma estrutura para uma função diretamente, podemos passar os componentes separadamente, ou passar um apontador para a estrutura. A primeira alternativa usa `dia_do_ano`, tal como a escrevemos no Capítulo 5:

```
d.dia_ano = dia_do_ano(d.ano, d.mes, d.dia);
```

A outra maneira é passar um apontador. Com a seguinte declaração de `contratacao`

```
struct data contratacao;
```

com a função `dia_do_ano` reescrita, podemos dizer

```
contratacao.dia_ano = dia_do_ano(&contratacao);
```

passando assim um apontador para `contratacao` para `dia_do_ano`. A função deve ser modificada porque seu argumento agora é um apontador ao invés de uma lista de variáveis.

```
dia_do_ano(ad) /* acha o dia do ano a partir de dia e mes */
{
    struct data *ad;
    {
        int i, dia, bissexto;

        dia = ad->dia;
        bissexto = ad->ano % 4 == 0 &&ad->ano % 100 != 0
            || ad->ano % 400 == 0;
        for (i = 1; i < ad->mes; i++)
            dia += tab_dia[bissexto][i];
        return (dia);
    }
}
```

A declaração

```
struct data *ad;
```

diz que `ad` é um apontador para uma estrutura do tipo `data`. A notação exemplificada por

```
ad->ano
```

é nova. Se `a` é um apontador para uma estrutura, então

```
a->membro-da-estrutura
```

refere-se ao membro particular. (O operador `->` é um sinal de menos seguido por `>`.)

Como `ad` aponta para a estrutura, o membro `ano` poderia ser referenciado usando-se

```
(*ad).ano
```

mas apontadores para estruturas são tão freqüentemente usados que a notação `->` é fornecida como uma abreviação conveniente. Os parênteses são necessários em `(*ad).ano` porque a precedência do operador `.` é maior que a de `*`. Ambos `->` e `.` são associativos da esquerda para a direita, de forma que

```
p->q->memb  
emp.nascimento.mes
```

são equivalentes a

```
(p->q)->memb  
(emp.nascimento).mes
```

Para completar, a nova função `dia_mes` segue, reescrita para usar a estrutura.

```
dia_mes(ad) /* acha mes e dia a partir do dia do ano */  
struct data *ad;  
{  
    int i, bissexto;  
  
    bissexto = ad->ano % 4 == 0 && ad->ano % 100 != 0  
              || ad->ano % 400 == 0;  
    ad->dia = ad->dia_ano;  
    for (i = 1; ad->dia > tab_dia[bissexto][i]; i++)  
        ad->dia -= tab_dia[bissexto][i];  
    ad->mes = i;  
}
```

Os operadores de estrutura `->` e `.`, juntamente com `()` para listas de argumentos e `[]` para indexação, estão no topo da hierarquia de precedência e portanto agrupam muito estreitamente. Por exemplo, dada a declaração

```
struct {  
    int x;  
    int *y;  
} *p;
```

então

```
++p->x
```

incrementa `x`, e não `p` porque a parentetização implícita é `++(p->x)`. Parênteses podem ser usados para alterar a ligação: `(++p)->x` incrementa `p` antes de acessar `x`, e `(p++)->x` incrementa `p` após acessar `x`. (Este último conjunto de parênteses é desnecessário. Por quê?)

Da mesma forma, `*p->y` busca o objeto para o qual `y` aponta; `*p->y++` incrementa `y` após acessar o objeto para o qual ele aponta (exatamente como `*s++`); `(*p->y)++` incrementa o objeto para o qual `y` aponta; e `*p++->y` incrementa `p` após acessar o objeto para o qual `y` aponta.

6.3 Arranjos de Estruturas

Estruturas são especialmente apropriadas para gerenciar arranjos de variáveis inter-relacionadas. Por exemplo, considere um programa para contar as ocorrências de cada palavra-chave de C. Precisamos de um arranjo de cadeias de caracteres para conter os nomes, e um arranjo de inteiros para os contadores. Uma possibilidade é usar dois arranjos paralelos `pal_chave` e `cont_chave`, como segue

```
char *pal_chave[NCHAVES];
int cont_chave[NCHAVES];
```

Mas, o fato dos arranjos serem paralelos indica que uma organização diferente é possível. A entrada para cada palavra-chave é, na realidade, um par:

```
char *pal_chave;
int cont_chave;
```

e há um arranjo de pares. A declaração da estrutura

```
struct chave {
    char *pal_chave;
    int cont_chave;
} tab_chave[NCHAVES];
```

define um arranjo `tab_chave` de estruturas do tipo `chave`, e aloca espaço para elas. Cada elemento do arranjo é uma estrutura. Poderíamos ter escrito:

```
struct chave {
    char *pal_chave;
    int cont_chave;
};
```

```
struct chave tab_chave[NCHAVES];
```

Desde que a estrutura `tab_chave` contém um conjunto constante de nomes, é mais fácil inicializá-la uma vez por todas durante a definição. A inicialização de estrutura é análoga às anteriores – a definição é seguida por uma lista de valores iniciais entre chaves;

```
struct chave {
    char *pal_chave;
    int cont_chave;
} tab_chave[] = {
    "break", 0,
    "case", 0,
    "char", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "while", 0
};
```

Os valores iniciais são listados em pares correspondentes aos membros da estrutura. Seria mais preciso colocarmos os valores iniciais de cada "linha" ou estrutura entre chaves, como segue


```

{ "break", 0 },
{ "case", 0 },
...

```

mas as chaves internas não são necessárias quando os valores iniciais são variáveis simples ou cadeias de caracteres, e quando estão todos presentes. Como já vimos, o compilador calculará o número de entradas no arranjo `tab_chave` se valores iniciais estiverem presentes e os colchetes `[ ]` são deixados vazios.

O programa de contagem de palavras-chaves começa com a definição de `tab_chave`. A rotina principal lê a entrada chamando repetidamente a função `lepal` que busca na entrada uma palavra por vez. Cada palavra é procurada em `tab_chave` com a versão da função de pesquisa binária que escrevemos no Capítulo 3. (Evidentemente, a lista de palavras-chaves deve ser dada em ordem crescente para que a pesquisa funcione.)

```

#define MAXPAL 20

main ( ) /* conta palavras-chaves de C */
{
    int n, t;
    char pal[MAXPAL];

    while ((t = lepal(pal, MAXPAL)) != EOF)
        if (t == LETRA)
            if ((n = pesq_binaria (pal, tab_chave, NCHAVES)) >= 0)
                tab_chave[n].cont_chave++;
    for (n = 0; n < NCHAVES; n++)
        if (tab_chave[n].cont_chave > 0)
            printf ("%4d %s\n",
                    tab_chave[n].cont_chave, tab_chave[n].pal_chave);
}

pesq_binaria (pal, tab, n) /* acha pal em tab[0] ... tab[n-1] */
char *pal;
struct chave tab [ ];
int n;
{
    int inicio, fim, meio, cond;

    inicio = 0;
    fim = n - 1;
    while (inicio <= fim) {
        meio = (inicio + fim) / 2;
        if ((cond = strcmp (pal, tab[meio].pal_chave)) < 0)
            fim = meio - 1;
        else if (cond > 0)
            inicio = meio + 1;
        else
            return (meio);
    }
}

```

```

    return (-1);
}

```

Mostraremos a função `lepal` em um momento; por ora, é suficiente dizer que ela retorna LETRA cada vez que ela encontra uma palavra, e copia a palavra no seu primeiro argumento.

A quantidade `NCHAVES` é o número de palavras-chaves em `tab_chave`. Embora pudéssemos contar as entradas manualmente, é muito mais fácil e seguro fazê-lo com a máquina, especialmente se a lista estiver sujeita a mudanças. Uma possibilidade seria terminar a lista de valores iniciais com um apontador nulo, e pesquisar `tab_chave` até encontrá-lo.

Mas isso não é necessário, já que o tamanho do arranjo pode ser determinado em tempo de compilação. O número de entrada é simplesmente

tamanho de tab_chave / tamanho de struct chave

C fornece um operador unário que funciona em tempo de compilação chamado `sizeof` que pode ser usado para calcular o tamanho de qualquer objeto. A expressão

`sizeof (objeto)`

produz um inteiro igual ao tamanho do objeto especificado (o tamanho é dado em unidades não especificadas chamadas bytes, que têm o mesmo tamanho de um `char`). O objeto pode ser uma variável, um arranjo, uma estrutura, ou o nome de um tipo básico como `int` ou `double`, ou o nome de um tipo derivado como uma estrutura. No nosso caso, o número de palavras-chaves é o tamanho do arranjo dividido pelo tamanho de um elemento do arranjo. Este cálculo é usado num comando `define` que atribui um valor a `NCHAVES`:

```
#define NCHAVES (sizeof (tab_chave) / sizeof (struct chave))
```

Agora vejamos a função `lepal`. Escrevemos `lepal` de uma forma mais geral que o necessário para este programa, mas ela não é muito mais complexa. `lepal` retorna a próxima "palavra" da entrada, onde uma palavra é uma cadeia de letras e dígitos começando com uma letra, ou uma única letra. O tipo do objeto é retornado como valor da função; o tipo é LETRA se o símbolo é uma palavra, EOF para sinalizar o fim de arquivo, ou o próprio caractere se este não é alfabético.

```

lepal (p, lim) /* le a proxima palavra da entrada */
char *p;
int lim;
{
    int c, t;

    if (tipo (c = *p++ = getch ()) != LETRA) {
        *p = '\0';
        return (c);
    }

    while (--lim > 0) {
        t = tipo (c = *p++ = getch ());
        if (t != LETRA && t != DIGITO) {
            ungetch (c);
            break;
        }
    }
}

```

```

    }
    *(p - 1) = '\0';
    return (LETRA);
}

```

lepal usa as rotinas `getch` e `ungetch` que escrevemos no Capítulo 4: quando pára a coleta de um símbolo alfabético, `lepal` terá obtido um caractere a mais do que devia. A chamada a `ungetch` devolve para a entrada o caractere obtido indevidamente.

`lepal` chama tipo para determinar o tipo de cada caractere da entrada. Segue uma versão aplicável apenas para o alfabeto ASCII:

```

tipo(c) /* retorna o tipo de um caractere ASCII */
int c;
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
        return (LETRA);
    else if (c >= '0' && c <= '9')
        return (DÍGITO);
    else
        return (c);
}

```

As constantes simbólicas `LETRA` e `DÍGITO` podem ter qualquer valor que não conflite com caracteres não-alfanuméricos e `EOF`; as escolhas óbvias são:

```

#define LETRA 'a'
#define DÍGITO '0'

```

`lepal` pode ser mais rápida se chamadas a uma função tipo forem substituídas por referências a um arranjo apropriado tipo []. A biblioteca padrão C fornece macros chamadas `isalpha` e `isdigit` que operam desta maneira.

Exercício 6-1. Faça esta modificação a `lepal` e observe a variação na velocidade de execução do programa.

Exercício 6-2. Escreva uma versão de tipo que seja independente do conjunto de caracteres.

Exercício 6-3. Escreva uma versão do programa de contagem de palavras-chaves que não conte ocorrências de palavras-chaves contidas em cadeias entre aspas.

6.4 Apontadores para Estruturas

Para ilustrarmos algumas das considerações envolvendo apontadores e arranjos de estruturas, vamos escrever novamente o programa de contagem de palavras-chaves, agora usando apontadores ao invés de índices de arranjos.

A declaração externa de `tab_chave` não precisa mudar, mas `main` e `pesq_binaria` devem mudar.

```

main ( ) /* conta palavras-chaves de C */
/* versao com apontadores */
{

```

```

int t;
char pal[MAXPAL];
struct chave *pesq_binaria ( ), *ap;

while ((t = lepal(pal, MAXPAL)) != EOF)
    if (t == LETRA)
        if ((ap=pesq_binaria(pal, tab_chave, NCHAVES)) != NULL)
            ap->cont_chave++;
        for (ap = tab_chave; ap < tab_chave + NCHAVES; ap++)
            if (ap->cont_chave > 0)
                printf ("%4d %s\n", ap->cont_chave, ap->pal_chave);
    }

struct chave *pesq_binaria (pal, tab, n) /* acha pal */
char *pal; /* em tab[0]... tab[n-1] */
struct chave tab [ ];
int n;
{
    int cond;
    struct chave *inicio = &tab[0];
    struct chave *fim = &tab[n-1];
    struct chave *meio;

    while (inicio <= fim) {
        meio = inicio + (fim-inicio) / 2;
        if ((cond = strcmp(pal, meio->pal_chave)) < 0)
            fim = meio - 1;
        else if (cond > 0)
            inicio = meio + 1;
        else
            return (meio);
    }
    return (NULL);
}

```

Há várias coisas a serem observadas aqui. Primeiro, a declaração de `pesq_binaria` deve indicar que ela retorna um apontador para uma estrutura do tipo `chave`, ao invés de um inteiro; isto é declarado em `main` e em `pesq_binaria`. Se `pesq_binaria` encontra a palavra, ela retorna um apontador para ela; caso contrário, retorna `NULL`.

Segundo, o acesso a elementos de `tab_chave` é feito por apontadores. Isto provoca uma mudança significativa em `pesq_binaria`: o cálculo do elemento médio não pode ser simplesmente

```
meio = (inicio + fim) / 2;
```

porque a *adição* de dois apontadores não produz um valor útil (mesmo quando dividido por 2), e, de fato, é ilegal. A construção deve ser mudada para

```
meio = inicio + (fim-inicio) / 2;
```

que atribui a `meio` o endereço do elemento intermediário entre `inicio` e `fim`.

Você deveria estudar também os valores iniciais de início e fim. É possível inicializar um apontador com o endereço de um objeto definido previamente; isto é precisamente o que fizemos aqui. Em *main*, escrevemos

```
for (ap = tab_chave; ap < tab_chave + NCHAVES; ap++)
```

Se *ap* é um apontador para uma estrutura, qualquer operação com *ap* leva em conta o tamanho da estrutura; portanto, *ap++* incrementa *ap* pelo valor correto para que o mesmo aponte para o próximo elemento do arranjo de estruturas. Porém não assumamos que o tamanho de uma estrutura seja a soma dos tamanhos dos seus membros — devido ao alinhamento necessário para objetos diferentes, pode haver “buracos” numa estrutura.

Finalmente, um aparte sobre o formato de um programa. Quando uma função retorna um tipo complicado como em

```
struct chave *pesq_binaria (pal, tab, n)
```

o nome da função pode se tornar difícil de ver, e de encontrar com o editor de texto. Um estilo alternativo é às vezes usado:

```
struct chave *  
pesq_binaria (pal, tab, n)
```

O formato a usar é uma questão de gosto; escolha a forma de que você mais goste e use-a consistentemente.

6.5 Estruturas Auto-Referenciadas

Suponha que desejemos atacar o problema mais geral de contar as ocorrências de *todas* as palavras em alguma entrada. Desde que a lista de palavras não é conhecida previamente, não podemos ordená-la convenientemente e usar uma pesquisa binária. Porém, não podemos fazer uma pesquisa linear para cada palavra que apareça para ver se ela já foi vista anteriormente, porque o programa não terminaria nunca. (Mais precisamente, o tempo de execução do programa cresceria proporcionalmente ao quadrado do número de palavras na entrada.) Como organizar os dados para obter eficiência com uma lista de palavras arbitrárias?

Uma solução é a de guardar o conjunto de palavras já vistas sempre ordenado, colocando cada palavra em sua posição apropriada assim que ela aparece. Isto não deve ser feito deslocando-se as palavras num arranjo linear, já que levaria muito tempo. Ao invés disto, usaremos uma estrutura de dados chamada de *árvore binária*.

A árvore contém um “nodo” para cada palavra distinta; cada nodo contém

```
um apontador para o texto da palavra  
um contador do número de ocorrências  
um apontador para o nodo filho esquerdo  
um apontador para o nodo filho direito
```

Nenhum nodo pode ter mais que dois filhos; ele *pode* ter apenas um ou nenhum.

Os nodos são mantidos de maneira que, em qualquer nodo, a subárvore à esquerda contenha somente palavras que são menores que a palavra do nodo, e a subárvore à direita, somente palavras que são maiores. Para verificar se uma nova palavra já está na árvore, parte-se da raiz e compara-se a nova palavra com a palavra do nodo. Se são iguais, a questão é respondida afirmativamente. Se a nova palavra é menor que a palavra do nodo, a pesquisa continua no filho esquerdo; caso contrário o filho direito é investigado. Se não há filho na direção requerida, a nova palavra não está na árvore, e o lugar do filho ausente

é onde ela deve ser inserida. Este processo de pesquisa é inerentemente recursivo, desde que a pesquisa a partir de qualquer nodo usa uma pesquisa a partir de um de seus filhos. Portanto, rotinas recursivas para a inserção e impressão são as mais naturais.

Voltemos à descrição de um nodo; ele é claramente uma estrutura com quatro componentes:

```
struct nodo {      /* o nodo basico */
    char *pal;      /* aponta para o texto */
    int  contador; /* numero de ocorrencias */
    struct nodo *esquerda; /* filho a esquerda */
    struct nodo *direita;  /* filho a direita */
};
```

Esta declaração "recursiva" de um nodo poderia parecer estranha, mas é correta. É ilegal para uma estrutura conter uma ocorrência de si própria, mas

```
struct nodo *esquerda;
```

declara *esquerda* como sendo um *apontador* para um nodo, e não um nodo.

O código do programa inteiro é surpreendentemente pequeno, dadas as várias rotinas de suporte que já escrevemos. São elas *lepal* para buscar a próxima palavra, e *aloca*, para fornecer espaço para conter as palavras lidas.

A rotina principal simplesmente lê palavras com *lepal* e as instala na árvore com *arvore*.

```
#define   MAXPAL   20

main ( )    /* contador de ocorrencias de palavras */
{
    struct nodo *raiz, *arvore ( );
    char pal[MAXPAL];
    int t;

    raiz = NULL;
    while ((t = lepal(pal, MAXPAL)) != EOF)
        if (t == LETRA)
            raiz = arvore(raiz, pal);
    impr_arvore(raiz);
}
```

arvore é simples. Uma palavra é apresentada por *main* ao nível mais alto (a raiz) da árvore. Em cada estágio, esta palavra é comparada com a palavra armazenada no nodo, e é enviada à subárvore esquerda ou direita por uma chamada recursiva a *arvore*. Eventualmente, a palavra casa-se com alguma já instalada na árvore (neste caso o contador é incrementado), ou um apontador nulo é encontrado, indicando que um nodo deve ser criado e acrescentado à árvore. Se um novo nodo é criado, *arvore* retorna um apontador para ele, que é instalado no nodo pai.

```
struct nodo *arvore(ap, p) /* coloca p em ou abaixo de ap */
struct nodo *ap;
char *p;
{
```

```

struct nodo *aloca_nodo ( );
char *guardacac ( );
int cond;

if (ap == NULL) { /* uma nova palavra */
    ap = aloca_nodo ( ); /* cria um novo nodo */
    ap->pal = guardacac (p);
    ap->contador = 1;
    ap->esquerda = ap->direita = NULL;
} else if ((cond = strcmp(p, ap->pal)) == 0)
    ap->contador ++; /* palavra repetida */
else if (cond < 0) /* menor vai na arvore a esquerda */
    ap->esquerda = arvore(ap->esquerda, p);
else /* maior vai na arvore a direita */
    ap->direita = arvore(ap->direita, p);
return (ap);
}

```

O armazenamento para o novo nodo é obtido pela rotina `aloca_nodo`, que é uma adaptação de `aloca` que escrevemos anteriormente. Ela retorna um apontador para um espaço livre capaz de conter um nodo da árvore. (Discutiremos este ponto em mais detalhes em um momento.) A nova palavra é copiada para um local escondido por `guardacac`, o contador é inicializado, e os dois filhos são inicializados com o valor nulo. Esta parte do código é executada apenas na borda da árvore, quando um novo nodo é acrescentado. Omitimos a verificação de erros (inadmissível num programa de produção) nos valores retornados por `guardacac` e `aloca_nodo`.

`impr_arvore` imprime a árvore partindo das subárvores à esquerda; em cada nodo, ela imprime a subárvore à esquerda (todas as palavras menores que a palavra atual), e então a palavra do nodo, depois a subárvore à direita (todas as palavras maiores). Se você se sentir inseguro sobre recursividade, desenhe uma árvore e imprima-a com `impr_arvore`; ela é uma das rotinas recursivas mais claras que você pode encontrar.

```

impr_arvore(ap) /* imprime a arvore ap recursivamente */
struct nodo *ap;
{
    if (ap != NULL) {
        impr_arvore(ap->esquerda);
        printf("%4d %s\n", ap->contador, ap->pal);
        impr_arvore(ap->direita);
    }
}

```

Uma observação prática: se a árvore se tornar “desbalanceada” devido às palavras não aparecerem em ordem randômica, o tempo de execução do programa pode crescer rapidamente. No pior caso, se as palavras já estão ordenadas, o programa faz uma desgastante simulação de pesquisa linear. Há generalizações da árvore binária, notadamente árvores 2-3 e árvores AVL, que não sofrem este comportamento no pior caso, mas não as descreveremos aqui.

Antes de deixarmos este exemplo, é interessante fazermos um breve comentário sobre um problema relacionado com alocadores de armazenamento. Evidentemente, é desejável que haja somente um alocador de armazenamento no programa, muito embora ele aloque diferentes tipos de objetos. Mas se um alocador tem de processar pedidos, digamos, para apontadores para `char` e apontadores para `struct nodo`, duas questões surgem. Primeiro, como atender à restrição da maioria das máquinas reais de que objetos de certos tipos devam satisfazer restrições de alinhamento (por exemplo, inteiros devem ser frequentemente alocados em endereços pares)? Segundo, que declarações podem fazer com que aloca retorne tipos diferentes de apontadores?

Restrições de alinhamento podem, em geral, ser satisfeitas facilmente, ao custo de algum espaço perdido, simplesmente assegurando que o alocador sempre retorne um apontador que satisfaça *todas* as restrições de alinhamento. Por exemplo, no PDP-11 é suficiente que aloca sempre retorne um apontador par, desde que qualquer tipo de objeto pode ser armazenado em endereços pares. O único custo é um caractere desperdiçado em pedidos de tamanho ímpar. Ações similares são tomadas em outras máquinas. Então, a implementação de aloca pode não ser transportável, mas seu uso o é. A função aloca do Capítulo 5 não garante qualquer alinhamento particular; no Capítulo 8, mostraremos como escrever esta função corretamente.

A questão da declaração de tipo para aloca é um problema para qualquer linguagem que faz verificação de tipos seriamente. Em C, o melhor procedimento é declarar que aloca retorna um apontador para `char`, e então explicitamente forçar o apontador para o tipo desejado. Isto é, se `p` é declarado como segue

```
char *p;
```

então

```
(struct nodo *) p
```

converte-o em um apontador do tipo `nodo` numa expressão. Então `aloca_nodo` é escrita como segue:

```
struct nodo *aloca_nodo( )
{
    char *aloca( );
    return ((struct nodo *) aloca(sizeof(struct nodo)));
}
```

Isto é mais do que é necessário para os compiladores atuais, mas representa uma atitude segura para o futuro.

Exercício 6-4. Escreva um programa que leia um programa C e imprima em ordem alfabética cada grupo de nomes de variáveis que são idênticas nos primeiros 7 caracteres, mas diferentes nos restantes. (Faça com que 7 seja um parâmetro.)

Exercício 6-5. Escreva um referenciador cruzado ("cross-referencer") básico: um programa que imprime uma lista de todas as palavras num documento, e, para cada palavra, uma lista dos números das linhas em que elas ocorrem.

Exercício 6-6. Escreva um programa que imprima as palavras distintas de sua entrada, ordenadas em ordem decrescente de frequência de ocorrência. Preceda cada palavra de seu contador.

6.6 Pesquisa em Tabela

Nesta seção escreveremos os princípios de um pacote de pesquisa em tabela como uma ilustração de aspectos adicionais de estruturas. Este código é tipicamente encontrado nas rotinas de gerenciamento de tabela de um macroprocessador ou de um compilador. Por exemplo, considere o comando C `#define`. Quando uma linha do tipo

```
#define SIM 1
```

é encontrada, o nome `SIM` e o texto de reposição `1` são armazenados numa tabela. Posteriormente, quando o nome `SIM` aparecer num comando tal como

```
empalavra = SIM;
```

ele deve ser trocado por `1`.

Há duas rotinas principais que manipulam os nomes e os textos de reposição. `instala(s, t)` registra o nome `s` e o texto de reposição `t` numa tabela; `s` e `t` são cadeias de caracteres. `pesquisa(s)` procura `s` na tabela, retornando um apontador para o local onde `s` se localiza caso seja encontrado, e `NULL` caso contrário.

O algoritmo usado é uma pesquisa "hash" — um nome é convertido num pequeno inteiro positivo, o qual é usado como índice num arranjo de apontadores. Um elemento do arranjo aponta para o início de uma lista encadeada de blocos cujos nomes têm o mesmo valor hash. Ele é `NULL` se nenhum nome gerou este valor hash.

Um bloco na lista encadeada é uma estrutura contendo apontadores para o nome, o texto de reposição, e o próximo bloco na lista. Um apontador `NULL` para o próximo bloco indica o fim da lista encadeada.

```
struct simb { /* entrada basica da tabela */
    char *nome;
    char *def;
    struct simb *proximo; /* proxima entrada na lista */
};
```

O arranjo de apontadores é simplesmente

```
#define TAMHASH 100
static struct simb *tabhash[TAMHASH]; /* tabela de apontadores */
```

A função de "hash", é usada por `instala` e `pesquisa`, e simplesmente adiciona os valores dos caracteres da cadeia e toma o resto da divisão desta soma pelo tamanho do arranjo. (Este não é o melhor algoritmo, mas tem o mérito de ser extremamente simples.)

```
hash(s) /* calcula o valor hash da cadeia s */
char *s;
{
    int valhash;
    for (valhash = 0; *s != '\0';)
        valhash += *s++;
    return(valhash % TAMHASH);
}
```

O processo de "hash" produz um índice inicial no arranjo `hashtab`; se a cadeia for encontrada em algum lugar, será na lista encadeada de blocos começando neste índice. A

pesquisa é feita por pesquisa. Se pesquisa encontrar a entrada correspondente, ela retorna um apontador para a entrada, caso contrário, ela retorna o valor NULL.

```
struct simb *pesquisa(s) /* procura s em tabhash */
char *s;
{
    struct simb *as;
    for (as = tabhash[hash(s)]; as != NULL; as = as->proximo)
        if (strcmp(s, as->nome) == 0)
            return (as); /* achou */
    return (NULL); /* nao achou */
}
```

instala usa pesquisa para determinar se o nome que está sendo instalado já está presente; se estiver, a nova definição substituirá a antiga. Se não, uma nova entrada é criada. instala retorna NULL se por alguma razão não houver espaço para a nova entrada.

```
struct simb *instala(nome, def) /* coloca (nome, def) */
char *nome, *def; /* em tabhash */
{
    struct simb *as, *pesquisa( );
    char *guardacad( ), *aloca( );
    int valhash;
    if ((as = pesquisa(nome)) == NULL) { /* nao achou */
        as = (struct simb *) aloca(sizeof(*as));
        if (as == NULL)
            return (NULL);
        if ((as->nome = guardacad(nome)) == NULL)
            return (NULL);
        valhash = hash(as->nome);
        as->proximo = tabhash[valhash];
        tabhash[valhash] = as;
    } else /* ja esta ai */
        libera(as->def); /* libera definicao antiga */
    if ((as->def = guardacad(def)) == NULL)
        return (NULL);
    return (as);
}
```

guardacad simplesmente copia a cadeia dada no seu argumento para um local seguro, obtido por uma chamada a aloca. Nós mostramos seu código no Capítulo 5. Já que chamadas a aloca e libera podem ocorrer em qualquer ordem, e já que há restrições de alinhamento, a versão simples de aloca do Capítulo 5 não é adequada; veja os Capítulos 7 e 8.

Exercício 6-7. Escreva uma rotina que remova um nome e uma definição da tabela mantida por instala e pesquisa.

Exercício 6-8. Implemente uma versão simples do processador de `#define` adequado para ser usado com programas C, baseada nas rotinas desta seção. `getch` e `ungetch` poderão ser úteis.

6.7 Campos

Quando o espaço de armazenamento é escasso, pode ser necessário se compactar vários objetos em uma única palavra da máquina; um uso especialmente comum é de um conjunto de sinalizadores de um só bit em aplicações tais como tabelas de símbolos de um compilador. Formatos de dados impostos externamente, tais como interfaces para dispositivos de hardware, requerem freqüentemente a habilidade de se manusear pedaços de uma palavra.

Imagine um fragmento de um compilador que manipula uma tabela de símbolos. Cada identificador de um programa tem certa informação a ele associada, por exemplo, se é uma palavra-chave ou não, se é ou não externo e/ou estático, e assim por diante. A forma mais compacta de codificar tais informações é um conjunto de sinalizadores de um só bit colocados num único char ou int.

A forma normal de implementar isso é de definir um conjunto de “máscaras” correspondentes às posições relevantes dos bits, como segue:

```
#define PAL_CHAVE 01
#define EXTERNO 02
#define ESTATICO 04
```

(Os números devem ser potências de dois.) O acesso aos bits usa então os operadores de deslocamento, mascaramento e complementação descritos no Capítulo 2.

Certas construções aparecem freqüentemente:

```
sinais |= EXTERNO | ESTATICO;
```

liga os bits que sinalizam EXTERNO e ESTATICO em sinais, enquanto

```
sinais &= ~(EXTERNO | ESTATICO);
```

desliga-os e

```
if ((sinais & (EXTERNO | ESTATICO)) == 0) ...
```

é verdadeiro se ambos os bits estão desligados.

Embora estas construções sejam facilmente dominadas, C oferece como alternativa a capacidade de definir e acessar campos dentro de uma palavra de forma direta ao invés de com operações lógicas em bits. Um *campo* é definido como sendo um conjunto de bits adjacentes dentro de um único int. A sintaxe de definição e acesso a campos é baseada em estruturas. Por exemplo, os `#define`'s da tabela de símbolos descritos acima poderiam ser mudados para a definição de três campos:

```
struct {
    unsigned eh_palchave : 1;
    unsigned eh_externo : 1;
    unsigned eh_estatico : 1;
} sinalizadores;
```

Isto define uma variável chamada `sinalizadores` que contém três campos de 1 bit cada. O número seguindo o dois-pontos representa o tamanho do campo em bits. Os campos são declarados `unsigned` para enfatizar que eles realmente são quantidades sem sinal.

Campos individuais são referenciados como `sinalizadores.eh_externo` etc., como quaisquer outros membros de estrutura. Campos comportam-se como pequenos inteiros sem sinal, e podem ser incluídos em expressões aritméticas como outros inteiros. Então os exemplos anteriores poderiam ser escritos mais naturalmente como segue:

```
sinalizadores.eh_externo = sinalizadores.eh_estatico = 1;
para ligar os bits;
sinalizadores.eh_externo = sinalizadores.eh_estatico = 0;
para desligá-los; e
if (sinalizadores.eh_externo == 0 && sinalizadores.eh_estatico == 0) ...
para testá-los.
```

Um campo não pode ultrapassar os limites de um `int`; se o tamanho de um campo causasse tal ultrapassagem, o mesmo é alinhado no início do próximo `int`. Campos não precisam ter nome; campos sem nomes (um dois-pontos e um tamanho somente) são usados para preenchimento. O tamanho especial 0 pode ser usado para forçar o alinhamento no início do próximo `int`.

Há um número de technicalidades que se aplicam a campos. A mais significativa talvez seja a de que campos são atribuídos da esquerda para a direita em algumas máquinas e da direita para a esquerda em outras, refletindo a natureza de hardware diferente. Isto significa que, embora campos sejam muito úteis para manter estruturas de dados definidas internamente, a questão de identificar o lado pelo qual começa o campo deve ser cuidadosamente considerada no acesso a dados definidos externamente.

Outras restrições devem ser lembradas: campos não têm sinal; eles só podem ser armazenados num `int` (ou num `unsigned`); eles não são arranjos; eles não têm endereços, de forma que o operador `&` não pode ser aplicado aos mesmos.

6.8 Uniões

Uma *união* é uma variável que pode conter (em tempos diferentes) objetos de tipos e tamanhos diferentes; o compilador trata das restrições de tamanho e alinhamento. Uniões fornecem um modo de se manipular tipos diferentes de dados numa única área de armazenamento, sem a necessidade de manter qualquer informação dependente de máquina no programa.

Novamente, como exemplo, tomemos uma tabela de símbolos de um compilador: suponha que constantes possam ser dos tipos `int`, `float` ou apontadores de caracteres. O valor de uma constante particular deve ser armazenado numa variável do tipo apropriado, ainda que seja mais conveniente para o gerenciamento da tabela que o valor ocupe a mesma quantidade de armazenamento e seja localizado no mesmo lugar, independentemente do tipo. Essa é a finalidade da união — permitir que uma única variável contenha qualquer um de vários tipos. Como no caso de campos, a sintaxe é baseada em estruturas.

```
union etiq_u {
    int vali;
    float valf;
    char *vala;
} valu;
```

A variável *valu* terá tamanho suficiente para conter o maior dos três tipos, independentemente da máquina na qual é compilada — o código é independente das características de hardware. Qualquer um desses tipos pode ser atribuído a *valu* e então ser usado em expressões, desde que seu uso seja consistente: o tipo recuperado deve ser o tipo mais recentemente armazenado na união; os resultados são dependentes de máquina se alguma coisa é armazenada como sendo de um tipo e recuperada como sendo de outro.

Sintaticamente, membros de uma união são acessados como:

nome-da-união.membro

ou

apontador-para-união->membro

exatamente como para estruturas. Se a variável *tipou* é usada para lembrar que tipo corrente está armazenado em *valu*, então poderíamos escrever:

```
if (tipou == INT)
    printf("%d\n", valu.vali);
else if (tipou == FLOAT)
    printf("%f\n", valu.valf);
else if (tipou == CADEIA)
    printf("%s\n", valu.vala);
else
    printf("tipo incorreto %d em tipou\n", tipou);
```

Unões podem ocorrer em estruturas e arranjos e vice-versa. A notação para acessar um membro de uma união numa estrutura (ou vice-versa) é idêntica à de estruturas aninhadas. Por exemplo, no arranjo de estruturas definido por

```
struct {
    char *nome;
    int  sinalizadores;
    int  tipou;
    union {
        int vali;
        float valf;
        char *vala;
    } valu;
} tabsimb [NSIMB];
```

a variável *vali* é referenciada como segue

tabsimb[i].valu.vali

e o primeiro caractere da cadeia *vala* por

**tabsimb[i].valu.vala*

Isto é, uma união é uma estrutura em que todos os membros têm deslocamento zero, a estrutura é grande o suficiente para conter o maior membro, e o alinhamento é apropriado para todos os tipos na união. Como no caso de estruturas, as únicas operações atualmente permitidas com uniões são o acesso a um membro e a obtenção do endereço; não se pode atribuir um valor a uma união, uniões não podem ser passadas para funções,

ou retornadas por funções. Apontadores para uniões podem ser usados de modo idêntico a apontadores para estruturas.

O alocador de armazenamento do Capítulo 8 mostra como uma união pode ser usada para forçar o alinhamento de uma variável num tipo particular de limite de armazenamento.

6.9 Typedef

C fornece uma facilidade chamada `typedef` para a criação de novos nomes de tipos de dados. Por exemplo, a declaração

```
typedef int TAMANHO;
```

torna o nome `TAMANHO` um sinónimo para `int`. O “tipo” `TAMANHO` pode ser usado em declarações, moldes etc., de modo idêntico ao tipo `int`:

```
TAMANHO tam, tamax;  
TAMANHO *tamanhos[ ];
```

De forma semelhante, a declaração

```
typedef char *CADEIA;
```

torna `CADEIA` um sinónimo para `char *` ou um apontador de caractere, que pode ser usado em declarações tais como

```
CADEIA p, alinha[LINHAS], aloca ( );
```

Observe que o tipo declarado num `typedef` aparece na posição de um nome de variável, e não à direita da palavra `typedef`. Sintaticamente, `typedef` é semelhante às classes de armazenamento `extern`, `static` etc. Também usamos letras maiúsculas para enfatizar os nomes.

Como um exemplo mais complicado, vamos usar `typedef` para os nodos de árvore mostrados anteriormente neste capítulo:

```
typedef struct nodo { /* o nodo basico */  
    char *pal; /* aponta para o texto */  
    int contador; /* numero de ocorrencias */  
    struct nodo *esquerda; /* filho a esquerda */  
    struct nodo *direita; /* filho a direita */  
} NODO, *AP_NODO;
```

Isto cria dois tipos de palavras-chaves chamadas `NODO` (uma estrutura) e `AP_NODO` (um apontador para a estrutura). A rotina `aloca_nodo` poderia ser escrita como segue:

```
AP_NODO aloca_nodo()  
{  
    char *aloca();  
    return((AP_NODO) aloca(sizeof(NODO)));  
}
```

Deve ser enfatizado que uma declaração `typedef` não cria um novo tipo; ela meramente acrescenta um novo nome para algum tipo existente. Nem há qualquer nova semântica; variáveis declaradas desta forma têm exatamente as mesmas propriedades das variáveis cujas declarações foram feitas explicitamente. Isto é, `typedef` é como um `# define` ex-

ceto que como `typedef` é interpretada pelo compilador, ela pode ser usada para a substituição de texto além da capacidade do préprocessador C. Por exemplo,

```
typedef int (*AFI) ( );
```

cria o tipo AFI ("apontador para uma função que retorna um `int`"), o qual pode ser usado em contexto tais como:

```
AFI strcmp, comprnum, troca;
```

no programa de ordenação do Capítulo 5.

Há duas razões principais para o uso de declarações `typedef`. A primeira é de parametrizar um programa para se proteger contra problemas de transportabilidade. Se `typedef` é usado para tipos de dados que podem ser dependentes de máquina, somente os `typedef`'s precisam ser mudados quando o programa é transportado. Uma situação comum é a de usar nomes `typedef` para várias quantidades inteiras, e então escolher um conjunto apropriado de `short`, `int` e `long` para cada máquina hospedeira.

A segunda finalidade de `typedef` é a de fornecer uma melhor documentação para um programa — um tipo chamado `AP_NODO` pode ser mais fácil de entender que uma declaração de um apontador para uma estrutura complicada.

Finalmente, há sempre a possibilidade de que no futuro o compilador, ou algum outro programa como o *lint*, possam fazer uso da informação contida em declarações `typedef` para implementar alguma verificação adicional de um programa.

Capítulo 7

ENTRADA E SAÍDA

Facilidades de entrada e saída não são partes integrantes da linguagem C, como enfatizado na nossa apresentação até agora. No entanto, programas reais interagem com seus ambientes de formas muito mais complexas que as vistas até agora. Neste capítulo, descreveremos a “biblioteca padrão de entrada e saída”, um conjunto de funções projetadas para fornecer um sistema padrão de entrada e saída para programas em C. As funções objetivam apresentar uma interface de programação conveniente, ainda que refletindo apenas operações que podem ser oferecidas na maioria dos sistemas operacionais modernos. As rotinas são eficientes o suficiente para que os usuários raramente tenham de evitá-las para obter “eficiência” independentemente das suas aplicações. Finalmente, as rotinas pretendem ser transportáveis, no sentido de que elas existem de forma compatível em qualquer sistema onde C exista, e que programas que restringem suas interações às facilidades fornecidas pela biblioteca padrão podem ser transportados de um sistema para outro essencialmente sem mudanças.

Não tentaremos descrever a biblioteca inteira de entrada e saída; estamos mais interessados em mostrar a essência da escrita de programas C que interagem com o ambiente fornecido pelo sistema operacional.

7.1 Acesso à Biblioteca Padrão

Cada arquivo-fonte que faz referência a uma função da biblioteca padrão deve conter a linha

```
#include <stdio.h>
```

no seu início. O arquivo `stdio.h` define certas macros e variáveis usadas pela biblioteca de entrada e saída. O uso de parênteses angulares `<e >` ao invés de aspas dirige o compilador a pesquisar o arquivo `stdio.h` num diretório contendo informações padrão (no UNIX, tipicamente é `/usr/include`).

Adicionalmente, pode ser necessário, quando da carga do programa, especificar a biblioteca explicitamente; por exemplo, no sistema UNIX do PDP-11, o comando para compilar um programa seria

```
cc arquivos-fontes, etc. -lS
```

onde `-lS` indica carga a partir da biblioteca padrão. (O caractere `l` é a letra “ele” minúscula).

7.2 Entrada e Saída Padrão – Getchar e Putchar

O mecanismo mais simples de entrada é ler um caractere por vez da “entrada padrão”, geralmente o terminal do usuário, com `getchar`. `getchar ( )` retorna o próximo caractere de entrada cada vez que é chamada. Na maioria dos ambientes que suportam C, o terminal pode ser substituído por um arquivo usando-se a convenção `<`: se um programa `prog` usa `getchar`, então o comando

```
prog < arqent
```

faz com que `prog` leia `arqent` ao invés do terminal. A comutação da entrada é feita de tal forma que `prog` não saiba o que aconteceu: em particular, a cadeia “`< arqent`” não é incluída nos argumentos da linha de comando em `argv`. A comutação de entrada é invisível também, se a entrada vier de outro programa através do mecanismo de “duto”; o comando

```
outroprog | prog
```

executa os programas `outroprog` e `prog`, e faz com que a entrada padrão de `prog` venha da saída padrão de `outroprog`.

`getchar` retorna o valor EOF quando ela encontra o fim do arquivo na sua entrada. A biblioteca padrão define a constante simbólica EOF com o valor `-1` (com um `#` define no arquivo `stdio.h`), mas testes devem ser feitos em termos de EOF, e não `-1`, para que sejam independentes do valor específico.

Para saída, `putchar(c)` coloca o caractere `c` na “saída padrão”, que também é o terminal do usuário, por default. A saída pode ser dirigida para um arquivo usando-se `>`: se `prog` usa `putchar`, a seguinte construção

```
prog > arqsaida
```

grava sua saída padrão em `arqsaida` ao invés do terminal. No sistema UNIX, um duto também pode ser usado:

```
prog | outroprog
```

coloca a saída padrão de `prog` na entrada padrão de `outroprog`. Novamente, `prog` não fica ciente do redirecionamento.

A saída produzida por `printf` também aparece na saída padrão de modo que chamadas a `putchar` e `printf` podem ser intercaladas.

Um número surpreendente de programas lê somente um fluxo de entrada e grava somente um fluxo de saída; para tais programas, a entrada/saída com `getchar`, `putchar` e `printf` pode ser inteiramente adequada e certamente é o suficiente no início. Isto é particularmente verdadeiro dadas as facilidades de redirecionamento e de dutos para conectar a saída de um programa à entrada de outro. Por exemplo, considere o programa *minuscúlo*, que mapeia sua entrada para minúsculo.

```
#include <stdio.h>

main ( )          /* converte entrada para minuscúlo */
{
    int c;

    while ((c = getchar ( )) != EOF)
```

```
    putchar(isupper(c) ? tolower(c) : c);
}
```

As “funções” `isupper` e `tolower` são, na realidade, macros definidas em `stdio.h`. A macro `isupper` testa se seu argumento é uma letra maiúscula, retornando um valor diferente de zero caso verdade, e zero caso contrário. A macro `tolower` converte uma letra maiúscula na minúscula correspondente. Independentemente de como essas macros são implementadas numa máquina particular, seu comportamento externo é o mesmo, e programas que as usam não têm de conhecer o conjunto de caracteres.

Para converter vários arquivos, você pode usar um programa como o utilitário `cat` do UNIX para agrupar os arquivos:

```
cat arq1 arq2 ... | minuscula > saida
```

e assim evitar aprender como acessar arquivos a partir de um programa. (*Cat* será apresentado posteriormente neste capítulo).

Na biblioteca padrão de entrada e saída, as “funções” `getchar` e `putchar` podem ser macros, evitando assim a sobrecarga de uma chamada de função para cada caractere lido. Mostraremos como isto é feito no Capítulo 8.

7.3 Saída Formatada — `printf`

As duas rotinas: `printf` para saída e `scanf` para entrada (próxima seção) permitem a tradução de quantidades numéricas de e para uma representação em caracteres. Elas permitem também a geração e interpretação de linhas formatadas. Nós já usamos `printf` informalmente em capítulos anteriores; segue uma descrição mais completa e precisa

```
printf (controle, arg1, arg2, ...)
```

`printf` converte, formata, e imprime seus argumentos na saída padrão sob o controle da cadeia controle. A cadeia de controle contém dois tipos de objetos: caracteres ordinários que são simplesmente copiados para o fluxo de saída, e especificações de conversão, cada uma causando a conversão e impressão do próximo argumento sucessivo de `printf`.

Cada especificação de conversão é introduzida pelo caractere `%` e encerrada com um caractere de conversão. Entre o `%` e o caractere de conversão, pode haver:

Um sinal de menos, que especifica ajustamento à esquerda do argumento convertido no seu campo.

Uma cadeia de dígitos especificando um tamanho mínimo de campo. O número convertido será impresso num campo, no mínimo, deste tamanho, e maior se necessário. Se o argumento convertido tem menos caracteres que o tamanho do campo, este é preenchido à esquerda (ou à direita, se o indicador de ajuste à esquerda está presente) para completar o tamanho do campo. O caractere de preenchimento é normalmente branco, e zero se o tamanho do campo foi especificado com um zero inicial (este zero não implica que o tamanho de campo seja octal).

Um ponto, o qual separa o tamanho do campo da próxima cadeia de dígitos.

Uma cadeia de dígitos (a precisão), que especifica o número máximo de caracteres de uma cadeia a serem impressos, ou o número de dígitos a serem impressos à direita do ponto decimal de um `float` ou `double`.

Um modificador de tamanho `l` (letra “ele” minúscula), o qual indica que o item de dados correspondente é um `long` ao invés de um `int`.

Os caracteres de conversão e seus significados são:

- d O argumento é convertido para a notação decimal.
- o O argumento é convertido para a notação octal sem sinal (sem um zero inicial).
- x O argumento é convertido para a notação hexadecimal sem sinal (sem um 0x inicial).
- u O argumento é convertido para a notação decimal sem sinal.
- c O argumento é um único caractere.
- s O argumento é uma cadeia; os caracteres da cadeia são impressos até que um caractere nulo seja encontrado ou até que o número de caracteres especificado na precisão seja atingido.
- e O argumento é tomado como um float ou double e convertido para notação decimal na forma `[−]m.nnnnnE[+−]xx` onde o tamanho da cadeia de n's é especificado pela precisão. A precisão default é 6.
- f O argumento é considerado como um float ou double e convertido para a notação decimal na forma `[−]mmm.nnnnn` o tamanho da cadeia de n's é especificado pela precisão. A precisão default é 6. Observe que a precisão não determina o número de dígitos significativos impressos no formato f.
- g Usar %e ou %f qualquer que seja menor; zeros não significativos não são impressos.

Se o caractere após o % não é um caractere de conversão, ele é impresso; assim % pode ser impresso com %%.

A maioria das conversões de formato é óbvia, e tem sido ilustrada nos capítulos anteriores. Uma exceção é a precisão relacionada com cadeias de caracteres. A tabela seguinte mostra o efeito de várias especificações na impressão de "Brasil, hoje" (12 caracteres). Colocamos dois-pontos em volta de cada campo para que você possa ver sua extensão.

: %10s:	: Brasil, hoje:
: %-10s:	: Brasil, hoje:
: %20s:	: Brasil, hoje:
: %-20s:	: Brasil, hoje :
: % 20,10s:	: Brasil, ho:
: %-20,10s:	: Brasil, ho :
: % .10s:	: Brasil, ho:

Uma advertência: printf usa seu primeiro argumento para decidir quantos argumentos seguem e quais são seus tipos. Ela ficará confusa, e você obterá respostas sem sentido, se não houver número suficiente de argumentos ou se eles não tiverem o tipo esperado.

Exercício 7-1. Escreva um programa que imprima uma entrada de uma forma compreensível. No mínimo, ele deve imprimir caracteres não-visíveis em octal ou hexadecimal (de acordo com preferências locais), e dobrar linhas longas.

7.4 Entrada Formatada – scanf

A função scanf é usada para entrada, como printf o é para saída, fornecendo as mesmas facilidades na direção oposta.

```
scanf (controle, arg1, arg2, ...)
```

`scanf` lê caracteres da entrada padrão, interpretando-os de acordo com o formato especificado em controle, e armazena os resultados nos argumentos restantes. O argumento de controle é descrito abaixo; os outros argumentos, *cada um dos quais deve ser um apontador*, indicam onde a entrada convertida correspondente deve ser armazenada.

A cadeia de controle contém normalmente especificações de conversão, que são usadas para controlar a interpretação das seqüências de entrada. A cadeia de controle pode conter:

Espaços, caracteres de tabulação e de nova-linha ("espaços em branco"), os quais são ignorados.

Caracteres ordinários (não %) que são esperados como os próximos caracteres diferentes de espaços em branco, no fluxo de entrada.

Especificações de conversão, consistindo do caractere %, um caractere de supressão de atribuição opcional *, um número opcional especificando um tamanho máximo de campo, e um caractere de conversão.

Uma especificação de conversão controla a conversão do próximo campo de entrada. Normalmente, o resultado é colocado na variável apontada pelo argumento correspondente. Se a supressão de atribuição é indicada pelo caractere *, entretanto, o campo de entrada é simplesmente saltado; nenhuma atribuição é feita. Um campo de entrada é definido como sendo uma cadeia de caracteres não-brancos; ele se estende até o próximo caractere branco ou até que o tamanho do campo seja alcançado, se houver. Isto implica que `scanf` lerá além do fim da linha para encontrar sua entrada, visto que o caractere de nova-linha é um "branco".

Os caracteres de conversão indicam a interpretação do campo de entrada; o argumento correspondente deve ser um apontador, tal como requerido pela semântica de chamada por valor de C. Os seguintes caracteres de conversão são aceitos:

- d Espera-se um inteiro decimal na entrada; o argumento correspondente deve ser um apontador para um inteiro.
- o Espera-se um inteiro octal (com ou sem um zero inicial) na entrada; o argumento correspondente deve ser um apontador para um inteiro.
- x Espera-se um inteiro hexadecimal (com ou sem um 0x inicial) na entrada; o argumento correspondente deve ser um apontador para um inteiro.
- h Espera-se um inteiro short na entrada; o argumento correspondente deve ser um apontador para um inteiro short.
- c Espera-se um único caractere na entrada; o argumento correspondente deve ser um apontador para um caractere; o próximo caractere de entrada é colocado no local indicado. O salto normal de caracteres em branco é suprimido neste caso; para ler o próximo caractere diferente de branco, use %1s.
- s Espera-se uma cadeia de caracteres na entrada; o argumento correspondente deve ser um apontador de caractere para um arranjo de caracteres de tamanho suficiente para aceitar a cadeia e um terminador \0 que será acrescido.
- f Espera-se um número de ponto flutuante na entrada; o argumento correspondente deve ser um apontador para um float. O caractere de conversão e é sinônimo de f. O formato de entrada para float é um sinal opcional, uma cadeia de dígitos possivelmente contendo um ponto decimal, e um campo de ex-

poente opcional contendo um E ou e seguido por um inteiro possivelmente com sinal.

Os caracteres de conversão d, o e x podem ser precedidos por l ("ele" minúsculo) para indicar que um apontador para um inteiro long, ao invés de int, aparece na lista de argumentos. De forma semelhante, os caracteres de conversão e ou f podem ser precedidos por l para indicar que um apontador para um double, ao invés de um float, está na lista de argumentos.

Por exemplo, a chamada

```
int i;
float x;
char nome[50];
scanf("%d %f %s", &i, &x, nome);
```

com a linha de entrada

```
25 54.32E-1 Thompson
```

atribuirá o valor 25 a i, o valor 5.432 a x, e a cadeia "Thompson", apropriadamente terminada por \0, será atribuída a nome. Os três campos de entrada podem ser separados por qualquer quantidade desejada de caracteres em brancos. A chamada

```
int i;
float x;
char nome[50];
scanf("%2d %f %*d %2s", &i, &x, nome);
```

com entrada

```
56789 0123 45a72
```

atribuirá 56 a i, 789.0 a x, saltará 0123, e colocará a cadeia "45" em nome. A próxima chamada a qualquer rotina de entrada começará a pesquisar na letra a. Nestes dois exemplos, nome é um apontador e não deve ser precedido de &.

Como outro exemplo, a calculadora elementar do Capítulo 4 pode ser escrita agora com scanf para fazer a conversão de entrada:

```
#include <stdio.h>

main ( )          /* calculadora elementar */
{
    double soma, v;

    soma = 0;
    while (scanf ("%lf", &v) != EOF)
        printf ("\t%.2f\n", soma += v);
}
```

scanf pára quando ela termina sua cadeia de controle, ou quando alguma entrada não casa com a especificação de controle. Ela retorna como seu valor o número de itens de entrada obtidos e atribuídos com sucesso. Isto pode ser usado para se decidir quantos itens de entrada foram encontrados. No fim do arquivo, EOF é retornado; observe que isto é diferente de zero, que significa que o próximo caractere de entrada não obedece à

primeira especificação na cadeia de controle. A próxima chamada a `scanf` continua a pesquisa imediatamente após o último caractere já retornado.

Uma advertência final: os argumentos para `scanf` *devem* ser apontadores. O erro mais comum é o de escrever:

```
scanf("%d", n);
```

ao invés de

```
scanf("%d", &n);
```

7.5 Conversão de Formato na Memória

As funções `scanf` e `printf` têm irmãos `sscanf` e `sprintf` que fazem as conversões correspondentes, mas operam numa cadeia ao invés de um arquivo. O formato geral é:

```
sscanf(cadeia, controle, arg1, arg2, ...)
```

```
sprintf(cadeia, controle, arg1, arg2, ...)
```

`sprintf` formata os argumentos `arg1`, `arg2`, etc., de acordo com controle como antes, mas coloca o resultado em cadeia ao invés da saída padrão. Evidentemente, cadeia deve ser grande o suficiente para conter o resultado. Como exemplo, se `nome` é um arranjo de caracteres e `n` é um inteiro, então

```
sprintf(nome, "temp%d", n);
```

cria uma cadeia da forma `tempnnn` em `nome`, onde `nnn` é o valor de `n`.

`sscanf` faz a conversão inversa — ela pesquisa cadeia de acordo com o formato em controle, e coloca os valores resultantes em `arg1`, `arg2`, etc. Estes argumentos devem ser apontadores. A chamada

```
sscanf(nome, "temp%d", &n)
```

atribui a `n` o valor da cadeia de dígitos seguindo `temp` em `nome`.

Exercício 7-2. Reescreva a calculadora de mesa do Capítulo 4 usando `scanf` e/ou `sscanf` para fazer a conversão de entrada e de números.

7.6 Acesso a Arquivos

Os programas escritos até agora têm lido da entrada padrão e gravado na saída padrão que assumimos como sendo predefinidas magicamente para o programa pelo sistema operacional local.

O próximo passo no assunto de entrada e saída é o de escrever um programa que acesse um arquivo que *não* esteja já conectado ao programa. Um programa que ilustra claramente a necessidade de tais operações é `cat` que concatena um conjunto de arquivos na saída padrão. `cat` é usado para imprimir arquivos no terminal, e como um coletor geral de entrada para programas que não têm a capacidade de acessar arquivos pelo nome. Por exemplo, o comando

```
cat x.c y.c
```

imprime o conteúdo dos arquivos `x.c` e `y.c` na saída padrão.

A questão é como fazer com que os arquivos indicados sejam lidos — isto é, como conectar os nomes externos que o usuário escolheu aos comandos que lêem os dados.

As regras são simples. Antes de poder ser lido ou gravado, um arquivo deve ser *aberto* pela função `fopen` da biblioteca padrão. `fopen` aceita um nome externo (como `x.c` ou `y.c`), executa algumas operações internas e negocia com o sistema operacional (detalhes não nos interessam aqui), e retorna um nome interno que deve ser usado em leituras ou gravações subsequentes no arquivo.

Este nome interno é, na realidade, um apontador, chamado um *apontador de arquivo*, para uma estrutura que contém informações sobre o arquivo tais como: localização de um buffer, a posição do caractere corrente no buffer, se o arquivo está sendo lido ou gravado etc. Os usuários não precisam conhecer os detalhes, porque parte das definições da biblioteca padrão obtidas de `stdio.h` é uma definição de uma estrutura chamada `FILE`. A única declaração necessária para um apontador de arquivo é exemplificada por:

```
FILE *fopen( ), *fp;
```

Isto diz que `fp` é um apontador para um `FILE` e `fopen` retorna um apontador para um `FILE`. Observe que `FILE` é um nome de um tipo, como `int`, e não uma etiqueta de estrutura; ele é implementado como um `typedef`. (Detalhes de como tudo isso funciona no UNIX são dados no Capítulo 8.)

A chamada atual de `fopen` num programa é

```
fp = fopen(nome, modo);
```

O primeiro argumento de `fopen` é o *nome* do arquivo, dado como uma cadeia de caracteres. O segundo argumento é o *modo*, também dado como uma cadeia de caracteres, a qual indica como vai se usar o arquivo. Modos permitidos são leitura ("`r`"), gravação ("`w`") ou adição ("`a`").

Se você abrir um arquivo que não existe para gravação ou adição, ele é criado (se possível). A abertura de um arquivo para gravação faz com que seu conteúdo antigo seja descartado. Uma tentativa de ler um arquivo que não existe é um erro, e pode também haver outras causas de erro (tais como tentar ler um arquivo para o qual não se tem permissão). Se há qualquer erro, `fopen` retorna o apontador nulo `NULL` (o qual por conveniência, é também definido em `stdio.h`).

A próxima coisa necessária é uma forma de ler ou gravar o arquivo uma vez que o mesmo está aberto. Há várias possibilidades, das quais `getc` e `putc` são as mais simples. `getc` retorna o próximo caractere de um arquivo; ela necessita do apontador de arquivo para informá-la qual arquivo ler. Então:

```
c = getc(fp);
```

coloca em `c` o próximo caractere do arquivo referenciado por `fp`, e EOF quando ela encontra o fim do arquivo. `putc` é a inversa de `getc`:

```
putc(c, fp);
```

coloca o caractere `c` no arquivo apontado por `fp` e retorna `c`. Como `getchar` e `putchar`, `getc` e `putc` podem ser macros ao invés de funções.

Quando um programa começa, três arquivos são abertos automaticamente, e apontadores são fornecidos para eles. Estes arquivos são a entrada padrão, a saída padrão e a saída padrão de erro; os apontadores correspondentes são chamados `stdin`, `stdout` e `stderr`. Normalmente eles estão todos conectados ao terminal, mas `stdin` e `stdout` podem ser redirecionados para arquivos ou dutos, tal como descrito na seção 7.2.

getchar e putchar podem ser definidos em termos de getc, putc, stdin e stdout como segue:

```
#define getchar( ) getc(stdin)
#define putchar(c) putc(c, stdout)
```

Para entrada ou saída formatada em arquivos, as funções fscanf e fprintf podem ser usadas. Elas são idênticas a scanf e printf, exceto que o primeiro argumento é um apontador de arquivo que especifica o arquivo a ser lido ou gravado; a cadeia de controle é o segundo argumento.

Com estas considerações preliminares, estamos em posição para escrever o programa *cat* para concatenar arquivos. O projeto básico é um que tem sido conveniente para muitos programas: se há argumentos da linha de comando, eles são processados em ordem. Senão, a entrada padrão é processada. Desta forma o programa pode ser usado sozinho ou como parte de um processo maior.

```
#include <stdio.h>

main(argc, argv)      /* cat: concatena arquivos */
int argc;
char *argv[ ];
{
    FILE *fp, *fopen( );
    if (argc == 1) /* nao tem arg.: copiar entrada padrao */
        copia_arq(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: nao pode abrir %s\n", *argv);
                break;
            } else {
                copia_arq(fp);
                fclose(fp);
            }
}

copia_arq(fp) /* copia arquivo fp na saida padrao */
FILE *fp;
{
    int c;

    while ((c = getc(fp)) != EOF)
        putc(c, stdout);
}
```

Os apontadores de arquivos, stdin e stdout são predefinidos na biblioteca de entrada e saída como a entrada e saída padrão, respectivamente; eles podem ser usados em qualquer lugar onde se possa usar um objeto do tipo FILE *. Eles são constantes, entretanto, e não variáveis; não tente atribuir nada a eles.

A função `fclose` é a inversa de `fopen`: ela encerra a conexão entre o apontador de arquivo e o nome externo, estabelecida por `fopen`, liberando o apontador para outro arquivo. Visto que a maioria dos sistemas operacionais tem algum limite no número de arquivos abertos simultaneamente para um programa, é uma boa idéia liberar as coisas quando elas não são mais necessárias como fizemos em `cat`. Há outra razão para usar o `fclose` num arquivo de saída — ele esvazia o buffer em que `putc` está colocando a saída. (`fclose` é chamada automaticamente para cada arquivo aberto quando um programa termina normalmente.)

7.7 Tratamento de Erro — `Stderr` e `Exit`

O tratamento de erros em `cat` não é ideal. O problema é que, se um dos arquivos não puder ser acessado por alguma razão, o diagnóstico é impresso no fim da saída concatenada. Isto é aceitável se a saída for para um terminal, mas é ruim se for para um arquivo, ou para outro programa através de um duto.

Para melhorar o tratamento, um segundo arquivo de saída, chamado `stderr`, é associado ao programa da mesma forma que `stdin` e `stdout`. Na medida do possível, saídas gravadas no `stderr` aparecem no terminal do usuário mesmo quando a saída padrão for redirecionada.

Vamos modificar `cat` para que grave suas mensagens de erro no arquivo de erro padrão.

```
#include <stdio.h>

main(argc, argv)    /* cat: concatena arquivos */
int argc;
char *argv[ ];
{
    FILE *fp, *fopen ( );

    if (argc == 1) /* nao tem arg.: copiar entrada padrao */
        copia_arq(stdin);

    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr,
                    "cat: nao pode abrir %s\n", *argv);
                exit(1);
            } else {
                copia_arq(fp);
                fclose(fp);
            }
        exit(0);
}
```

O programa sinaliza erros de duas formas. Os diagnósticos de saída produzidos por `fprintf` vão para `stderr`, impressos no terminal do usuário ao invés de desaparecer num duto ou arquivo de saída.

O programa inclui também a função `exit` da biblioteca padrão, que termina a execução do programa quando é chamada. O argumento de `exit` está disponível a qualquer processo que o tenha chamado, e assim o sucesso ou falha do programa podem ser testados por outro programa que o use como um subprocesso. Por convenção, um valor de retorno igual a 0 sinaliza que tudo está bem, e vários valores diferentes de zero sinalizam situações anormais.

`exit` chama `fclose` para cada arquivo de saída aberto, para esvaziar qualquer saída ainda num buffer, e chama então uma rotina chamada `_exit`. A função `_exit` causa o término imediato sem qualquer esvaziamento de buffers; evidentemente, ela pode ser chamada diretamente se desejado.

7.8 Entrada e Saída de Linhas

A biblioteca padrão fornece uma rotina `fgets` que é semelhante à função `getline` que nós usamos por todo livro. A chamada

```
fgets(linha, MAXLINHA, fp)
```

lê a próxima linha de entrada (incluindo o caractere de nova-linha) do arquivo `fp` para o arranjo de caracteres `linha`; no máximo `MAXLINHA-1` caracteres serão lidos. A linha resultante é terminada com `\0`. Normalmente `fgets` retorna `linha`; no fim do arquivo, ela retorna `NULL`. (`getline` retorna o tamanho da linha, e zero para sinalizar o fim do arquivo.)

Para a saída, a função `fputs` grava uma cadeia (que não precisa conter um caractere de nova-linha) em um arquivo:

```
fputs(linha, fp)
```

Para mostrar que não há nada mágico sobre funções tais como `fgets` e `fputs`, aqui estão elas, copiadas diretamente da biblioteca padrão de entrada e saída:

```
#include <stdio.h>
```

```
char *fgets(s, n, iop) /* obtém ate n caracteres de iop */
char *s;
int n;
register FILE *iop;
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return ((c == EOF && cs == s) ? NULL : s);
}
```

```
fputs(s, iop) /* gravar a cadeia s no arquivo iop */
register char *s;
register FILE *iop;
```

```

{
    register int c;

    while (c = *s++)
        putc(c, iop);
}

```

Exercício 7-3. Escreva um programa que compare dois arquivos, imprimindo a primeira linha e a posição do caractere onde eles diferem.

Exercício 7-4. Modifique o programa de pesquisa de padrão do Capítulo 5 para aceitar sua entrada a partir de um conjunto de arquivos, ou se nenhum arquivo for dado como argumento, a partir da entrada padrão. Seria bom imprimir o nome do arquivo quando uma linha com o padrão especificado fosse encontrada?

Exercício 7-5. Escreva um programa que imprima um conjunto de arquivos, iniciando cada um numa nova página, com um título e um contador de página para cada arquivo.

7.9 Algumas Funções Diversas

A biblioteca padrão provê uma variedade de funções, algumas das quais são especialmente úteis. Nós já mencionamos as funções `strlen`, `strcpy`, `strcat` e `strcmp`. Seguem algumas outras.

Teste e Conversão de Classe de Caracteres

Várias macros oferecem testes e conversões de caracteres:

<code>isalpha(c)</code>	não zero se <code>c</code> é alfabético, 0 caso contrário
<code>isupper(c)</code>	não zero se <code>c</code> é maiúscula, 0 caso contrário
<code>islower(c)</code>	não zero se <code>c</code> é minúscula, 0 caso contrário
<code>isdigit(c)</code>	não zero se <code>c</code> é dígito, 0 caso contrário
<code>isspace(c)</code>	não zero se <code>c</code> é branco, tabulação ou nova-linha, 0 caso contrário
<code>toupper(c)</code>	converte <code>c</code> para maiúsculo
<code>tolower(c)</code>	converte <code>c</code> para minúsculo

Ungetc

A biblioteca padrão fornece uma versão mais restrita da função `ungetch` que escrevemos no Capítulo 4; ela é chamada `ungetc`.

```
ungetc(c, fp)
```

devolve o caractere `c` para o arquivo `fp`. Apenas um caractere devolvido é permitido por arquivo. `ungetc` pode ser usada com qualquer função de entrada e macros tais como `scanf`, `getc` ou `getchar`.

Chamada System

A função `system(s)` executa o comando contido na cadeia de caracteres `s`, e então retorna à execução do programa corrente. O conteúdo de `s` depende do sistema operacio-

nal local. Como um exemplo trivial no UNIX, a linha

```
system("date");
```

faz com que o programa `date` seja executado; ele imprime a data e a hora correntes.

Gerenciamento de Memória

A função `calloc` é parecida com a aloca que usamos em capítulos anteriores.

```
calloc(n, sizeof(objeto));
```

retorna um apontador para `n` objetos do tamanho especificado, ou `NULL` se o pedido não pode ser satisfeito. O armazenamento é inicializado com o valor zero.

O apontador tem alinhamento apropriado para o objeto em questão, mas ele deve ser forçado a ter o tipo apropriado, como segue:

```
char *calloc( );  
int *ip;
```

```
ip = (int *) calloc(n, sizeof(int));
```

`cfree(p)` libera o espaço apontado por `p`, onde `p` foi obtido por uma chamada a `calloc`. Não há restrições na ordem em que o espaço é liberado, mas é um erro sério liberar alguma coisa não obtida com `calloc`.

O Capítulo 8 mostra a implementação de um alocador de memória tal como `calloc`, onde os blocos alocados podem ser liberados em qualquer ordem.

Capítulo 8

A INTERFACE COM O SISTEMA UNIX *

O material deste capítulo diz respeito à interface entre programas C e o sistema operacional UNIX**. Visto que a maioria dos usuários de C usa sistemas UNIX, o material será útil para a maioria dos usuários. Mesmo que você use C numa máquina diferente, entretanto, você deverá atingir uma maior compreensão da programação C pelo estudo dos exemplos.

O capítulo está dividido em três áreas principais: a entrada e saída, sistema de arquivos, e um alocador de memória. As duas primeiras partes assumem alguma familiaridade com as características externas do UNIX.

O Capítulo 7 lidou com uma interface de sistema uniforme numa variedade de sistemas. Em qualquer sistema particular, as rotinas da biblioteca padrão têm de ser escritas em termos das facilidades de entrada e saída atualmente disponíveis no sistema hospedeiro. Nas próximas poucas seções nós descreveremos as chamadas básicas ao sistema para entrada e saída no sistema operacional UNIX, e ilustraremos como partes da biblioteca padrão podem ser implementadas com elas.

8.1 Descritores de Arquivos

No sistema operacional UNIX, toda entrada e saída é feita pela leitura ou gravação de arquivos, porque todos os dispositivos periféricos, até o terminal do usuário, são arquivos no sistema de arquivos. Isto significa que uma única interface simples e homogênea controla toda a comunicação entre um programa e dispositivos periféricos.

No caso mais geral, antes de ler e gravar um arquivo, é necessário informar o sistema de sua intenção, num processo chamado de "abertura" do arquivo. Se você quer gravar no arquivo, pode ser necessário criá-lo. O sistema verifica a legalidade do pedido (o arquivo existe? Você tem permissão para acessá-lo?), e, se tudo estiver certo, retorna ao programa um pequeno inteiro positivo chamado um *descritor de arquivo*. Sempre que uma entrada ou saída for feita no arquivo, o descritor de arquivo é usado ao invés do nome para identificá-lo. (Isto é essencialmente análogo ao uso de READ(5, ...) e WRITE(6, ...) no Fortran.) Toda informação sobre um arquivo aberto é mantida pelo sistema; o programa do usuário se refere ao arquivo somente pelo descritor de arquivo.

* Vários nomes foram deixados em inglês neste capítulo para manter compatibilidade com o sistema operacional UNIX (N. T.)

** UNIX é uma marca registrada dos Bell Laboratories.

Já que a entrada e saída envolvendo o terminal do usuário é muito comum, um procedimento especial torna isso mais conveniente. Quando o interpretador de comando (o shell) executa um programa, ele abre três arquivos, com descritores 0, 1, 2, chamados a entrada padrão, a saída padrão, e a saída padrão de erro. Todos eles estão normalmente conectados ao terminal, de forma que se um programa lê o descritor de arquivo 0 e grava os descritores de arquivos 1 e 2, ele pode fazer entrada e saída no terminal sem se preocupar com a abertura de arquivos.

O usuário de um programa pode *redirecionar* a entrada e saída, de ou para arquivos com < e >:

```
prog < arqent > arqsaida
```

Neste caso, o shell muda as atribuições padrão dos descritores 0 e 1 do terminal para os arquivos indicados. Normalmente, o descritor 2 permanece conectado ao terminal, de forma que mensagens de erro possam ir para ele. Observações similares valem se a entrada ou saída for associada com um duto. Deve-se observar que a atribuição de arquivos são sempre alteradas pelo shell, e não pelo programa. O programa não sabe de onde sua entrada vem ou para onde sua saída vai, desde que ele use o descritor 0 para entrada e 1 e 2 para saída.

8.2 Entrada e Saída de Baixo Nível – Read e Write

O nível mais baixo de entrada e saída no UNIX não prevê buffers ou qualquer outro serviço; ele é de fato uma entrada direta no sistema operacional. Toda entrada e saída é feita por duas funções chamadas read e write. Para ambas, o primeiro argumento é um descritor de arquivo. O segundo argumento é um buffer do seu programa, de onde os dados vêm ou para onde vão. O terceiro argumento é o número de bytes a serem transferidos. As chamadas são:

```
n_lidos = read(da, buf, n);
```

```
n_gravados = write(da, buf, n);
```

Cada chamada retorna um contador de bytes que é o número de bytes transferidos. Na leitura, o número de bytes retornados pode ser menor que o número pedido. Um valor de retorno zero indica o fim do arquivo, e -1 indica um erro de algum tipo. Na gravação, o valor retornado é o número de bytes gravados; é geralmente um erro se ele não for igual ao número pedido.

O número de bytes a serem lidos ou gravados é arbitrário. Os valores mais comuns são 1, que significa um caractere por vez (sem buffer), e 512, que corresponde ao tamanho do bloco físico em muitos dispositivos periféricos. Este último tamanho será o mais eficiente, mas mesmo um caractere por vez não é extraordinariamente custoso.

Juntando esses fatos, podemos escrever um programa simples para copiar sua entrada na sua saída, o equivalente do programa de cópia de arquivo escrito no Capítulo 1. No UNIX, este programa copiará qualquer coisa para qualquer coisa, visto que a entrada e a saída podem ser redirecionadas para qualquer arquivo ou dispositivo.

```
#define TAMBUF 512 /* melhor tamanho para UNIX no PDP-11 */

main( )          /* copia a entrada na saída */
{
```

```

char buf[TAMBUF];
int n;

while ((n = read(0, buf, TAMBUF)) > 0)
    write(1, buf, n);
}

```

Se o tamanho do arquivo não for um múltiplo de TAMBUF, algum read retornará um número menor de bytes a serem gravados por write; a próxima chamada a read retornará zero.

É instrutivo ver como read e write podem ser usados para construir rotinas de mais alto nível como getchar, putchar etc. Por exemplo, segue uma versão de getchar que faz entrada sem buffer.

```

#define MASCARA 0377 /* para forçar caract. a serem > 0 */

getchar( ) /* entrada caractere a caractere, sem buffer */
{
    char c;

    return ((read(0, &c, 1) > 0) ? c & MASCARA : EOF);
}

```

c deve ser declarado como sendo do tipo char, porque read aceita um apontador de caracteres. O caractere retornado deve ser mascarado com o valor 0377 para assegurar que ele seja positivo; caso contrário, a extensão de sinal pode torná-lo negativo. (A constante 0377 é apropriada para o PDP-11 mas não necessariamente para outras máquinas.)

A segunda versão de getchar faz entrada em grandes blocos, e devolve um caractere por vez.

```

#define MASCARA 0377 /* para forçar caract. a serem > 0 */
#define TAMBUF 512

getchar( ) /* versão com buffer */
{
    static char buf[TAMBUF];
    static char *abuf = buf;
    static int n = 0;

    if (n == 0) { /* buffer esta vazio */
        n = read(0, buf, TAMBUF);
        abuf = buf;
    }

    return ((--n >= 0) ? *abuf++ & MASCARA : EOF);
}

```

8.3 Open, Creat, Close, Unlink

Para ler ou gravar outros arquivos além da entrada, saída e saída padrão de erro,

você deve abri-los de forma explícita. Há dois pontos de entrada no sistema para isto, `open` e `creat` [sic].

`open` é semelhante a `fopen` discutida no Capítulo 7, exceto que ao invés de um apontador, ele retorna um descritor de arquivo, que é um `int`.

```
int fd;
```

```
fd = open(nome, modo);
```

Como no caso de `fopen`, o argumento `nome` é uma cadeia de caracteres correspondendo ao nome externo do arquivo. O argumento de modo de acesso é, entretanto, diferente: modo tem valor 0 para leitura, 1 para gravação e 2 para leitura e gravação. `open` retorna -1 se qualquer erro ocorrer; caso contrário ele retorna um descritor de arquivo válido.

É um erro tentar abrir um arquivo inexistente. O ponto de entrada `creat` é usado para criar arquivos novos, ou regravar arquivos velhos.

```
fd = creat(nome, modop);
```

retorna um descritor de arquivo se foi possível criar o arquivo `nome`, e -1 caso contrário. Se o arquivo já existe, `creat` esvaziá-lo-á; não é um erro criar um arquivo já existente.

Se o arquivo é novo, ele é criado com *modo de proteção* especificado pelo argumento `modop`. No sistema de arquivo UNIX, há nove bits de proteção associados com o arquivo, controlando a permissão de leitura, gravação e execução para o dono do arquivo, para o grupo do dono, e para todos os outros usuários. Então, um número octal de três dígitos é a forma mais conveniente de especificar as permissões. Por exemplo, 0755 especifica permissão de leitura, gravação e execução para o dono, e leitura e execução para o grupo e para todos os outros usuários.

Como ilustração, segue uma versão simplificada do utilitário `cp` presente no UNIX, um programa que copia um arquivo em outro. (A simplificação principal é que nossa versão copia somente um arquivo, e não permite que o segundo argumento seja um diretório.)

```
#define NULL 0
#define TAMBUF 512
#define MODOP 0644    /* LG para dono, L para o grupo e os outros */

main(argc, argv)    /* cp: copia f1 em f2 */
int argc;
char *argv[ ];
{
    int f1, f2, n;
    char buf[TAMBUF];

    if (argc != 3)
        erro("Sintaxe: cp de para", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        erro("cp: nao pode abrir %s", argv[1]);
    if ((f2 = creat(argv[2], MODOP)) == -1)
        erro("cp: nao pode criar %s", argv[2]);

    while ((n = read(f1, buf, TAMBUF)) > 0)
```



```

        if (write(f2, buf, n) != n)
            erro("cp: erro de gravacao", NULL);
    exit (0);
}

erro(s1, s2)      /* imprime mensagem de erro e morre */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}

```

Há um limite (tipicamente 15 a 25) no número de arquivos que um programa pode manter abertos simultaneamente. Assim, qualquer programa que pretende processar muitos arquivos deve estar preparado para reusar descritores de arquivos. A rotina `close` desfaz a conexão entre um descritor de arquivo e um arquivo aberto, e libera o descritor para ser usado com algum outro arquivo. O término de um programa com `exit`, ou pelo retorno a partir do programa principal fecha todos os arquivos abertos.

A função `unlink(nome)` remove o arquivo `nome` do sistema de arquivos.

Exercício 8-1. Reescreva o programa `cat` do Capítulo 7 usando `read`, `write`, `open` e `close` ao invés dos seus equivalentes da biblioteca padrão. Experimente para determinar as velocidades relativas das duas versões.

8.4 Acesso Randômico – `Seek` e `Lseek`

A entrada e saída de arquivos é normalmente sequencial: cada `read` ou `write` acessa a posição seguinte à anterior. Quando necessário, entretanto, um arquivo pode ser lido ou gravado em qualquer ordem arbitrária. A chamada do sistema `lseek` fornece uma maneira de se posicionar dentro de um arquivo sem leitura ou gravação:

```
lseek(da, deslocamento, origem);
```

força a posição corrente do arquivo cujo descritor é `da`, a se mover para a posição deslocamento, o qual é relativo à posição especificada por origem. Leituras ou gravações subsequentes começarão nesta posição. deslocamento é um `long`; `da` e origem são do tipo `int`. origem pode ser 0, 1, ou 2 para especificar que deslocamento deve ser medido a partir do início, da posição corrente, ou do final do arquivo, respectivamente. Por exemplo, para adicionar informação a um arquivo, posicione-se no fim do arquivo antes de gravar:

```
lseek(da, 0L, 2);
```

Para voltar para o início (*rewind*),

```
lseek(da, 0L, 0);
```

Observe o argumento 0L; ele poderia ser também escrito como `(long) 0`.

Com `lseek`, é possível tratarmos arquivos mais ou menos como se fossem grandes arranjos, ao custo de um acesso mais lento. Por exemplo, a seguinte função lê qualquer número de bytes a partir de qualquer lugar arbitrário de um arquivo.

```
le(da, pos, buf, n)      /* le n bytes da posicao pos */
int da, n;
```

```

long pos;
char *buf;
{
    lseek(da, pos, 0); /* posicione-se */
    return(read(da, buf, n));
}

```

Em versões do UNIX anteriores à versão 7, o ponto de entrada básico é chamado seek. seek é idêntico a lseek, exceto que o argumento deslocamento é um int ao invés de um long. Visto que inteiros no PDP-11 têm somente 16 bits, o deslocamento especificado para seek é limitado a 65.535; por esta razão, os valores 3, 4, 5 para origem fazem com que seek multiplique o deslocamento dado por 512 (o número de bytes num bloco físico) para então interpretar origem como 0, 1 ou 2, respectivamente. Assim o posicionamento num lugar arbitrário de um arquivo grande requer dois seek, um para selecionar o bloco, e um com origem igual a 1 para obter o byte desejado dentro do bloco.

Exercício 8-2. Evidentemente, a função seek pode ser escrita em termos de lseek e vice-versa. Escreva cada uma em termos da outra.

8.5 Exemplo – Uma Implementação de Fopen e Getc

Vamos ilustrar a função de algumas dessas peças mostrando uma implementação das rotinas fopen e getc da biblioteca padrão de entrada e saída.

Lembre-se que arquivos na biblioteca padrão são descritos por apontadores ao invés de descritores de arquivos. Um apontador de arquivo aponta para uma estrutura que contém várias peças de informação sobre o arquivo: um apontador para um buffer, para que o arquivo possa ser lido em grandes pedaços; um contador do número de caracteres presentes no buffer; um apontador para a próxima posição de caractere no buffer; alguns sinalizadores que descrevem o modo de leitura/gravação etc.; e o descritor de arquivo.

A estrutura de dados que descreve um arquivo está contida no arquivo stdio.h que deve ser incluído (usando #include) em qualquer arquivo-fonte que use as rotinas da biblioteca padrão. Ele é incluído também por funções desta biblioteca. Na seguinte porção de stdio.h, nomes que devem ser usados somente pelas funções da biblioteca começam com um caractere sublinhado para que tenham menos probabilidade de entrar em conflito com nomes no programa do usuário.

```

#define _BUFSIZE 512
#define _NFILE 20 /* numero. de arquivos possiveis */

typedef struct _iobuf {
    char *_ptr; /* posicao do proximo caractere */
    int _cnt; /* numero de caracteres no buffer */
    char *_base; /* localizacao do buffer */
    int _flag; /* modo de acesso ao arquivo */
    int _fd; /* descritor do arquivo */
} FILE;
extern FILE _job[_NFILE];

#define stdin (&_job[0])

```

```

#define stdout (&_iob[1])
#define stderr (&_iob[2])

#define _READ      01 /* arquivo aberto para leitura */
#define _WRITE     02 /* arquivo aberto para gravacao */
#define _UNBUF     04 /* arquivo nao tem buffer */
#define _BIGBUF    010 /* buffer grande alocado */
#define _EOF 020 /* EOF alcançado neste arquivo */
#define _ERR 040 /* erro ocorreu neste arquivo */
#define NULL 0
#define EOF (-1)

#define getc(p)    (--(p)->_cnt)>=0 \
    ? *(p)->_ptr++ & 0377 : _fillbuf(p)
#define getchar() getc(stdin)

#define putc(x,p)  (--(p)->_cnt)>=0 \
    ? *(p)->_ptr++ = (x) : _flushbuf((x),p)
#define putchar(x) putc(x,stdout)

```

A macro `getc` normalmente decrementa o contador, avança o apontador e retorna o caractere (um `#define` longo é continuado com uma contra-barra). Se o contador se torna negativo, entretanto, `getc` chama a função `_fillbuf` para preencher o buffer, reinicializar o conteúdo da estrutura, e retornar um caractere. Uma função pode apresentar uma interface transportável, ainda que ela própria contenha construções não-transportáveis: `getc` mascara o caractere com o valor 0377, que anula a extensão de sinal feita pelo PDP-11 e garante que todos os caracteres sejam positivos.

Embora não discutamos quaisquer detalhes, incluímos a definição de `putc` para mostrar que ela opera de forma muito semelhante a `getc`, chamando uma função `_flushbuf` quando o buffer está cheio.

A função `fopen` pode ser escrita agora. A maior parte de `fopen` tem a ver com a abertura do arquivo e com o posicionamento no local correto, e com a atualização dos sinalizadores para indicar o estado correto. `fopen` não aloca qualquer espaço para o buffer; isto é feito por `_fillbuf` quando o arquivo é lido pela primeira vez.

```

#include <stdio.h>
#define MODOP 0644 /* LG para dono; L para outros */

FILE *fopen(nome, modo) /* abre arquivo, retorna um apontador */
register char *nome, *modo;
{
    register int da;
    register FILE *aa;

    if (*modo != 'r' && *modo != 'w' && *modo != 'a') {
        fprintf(stderr, "modo %s ilegal na abertura de %s\n",
            modo, nome);
        exit(1);
    }
}

```

```

for (aa = _iob; aa < _iob + _NFILE; aa++)
    if ((aa->_flag & (_READ | _WRITE)) == 0)
        break; /* achou uma entrada livre */
if ((aa >= _iob + _NFILE) /* nao tem entrada livre */)
    return(NULL);

if (*modo == 'w') /* acessa o arquivo */
    da = creat(nome, MODOP);
else if (*modo == 'a') {
    if ((fd = open(nome, 1)) == -1)
        da = creat(nome, MODOP);
    lseek(da, 0L, 2);
} else
    fd = open(nome, 0);
if (da == -1) /* nao pode acessar o arquivo nome */
    return(NULL);

aa->_fd = da;
aa->_cnt = 0;
aa->_base = NULL;
aa->_flag &= ~(_READ | _WRITE);
aa->_flag |= (*modo == 'r') ? _READ : _WRITE;
return(aa);
}

```

A função `_fillbuf` é um pouco mais complicada. A complexidade maior está no fato de que `_fillbuf` tenta permitir o acesso ao arquivo mesmo quando não há espaço suficiente de memória para fazer armazenamento de entrada e saída. Se o espaço para o buffer pode ser obtido por `calloc`, tudo está bem; se não, `_fillbuf` faz entrada e saída sem buffer usando um único caractere armazenado num arranjo privado.

```
#include <stdio.h>
```

```

_fillbuf(aa) /* aloca e enche o buffer de entrada */
register FILE *aa;
{
    static char smallbuf[_NFILE]; /* para E/S sem buffer */
    char *calloc();

    if ((aa->_flag & _READ) == 0 || (aa->_flag & (_EOF | _ERR)) != 0)
        return(Eof);
    while (aa->_base == NULL) /* acha buffer */
        if (aa->_flag & _UNBUF) /* sem buffer */
            aa->_base = &smallbuf[aa->_fd];
        else if ((aa->_base = calloc(_BUFSIZE, 1)) == NULL)
            aa->_flag |= _UNBUF; /* nao pode achar buf */
        else
            aa->_flag |= _BIGBUF; /* achou um grande */
    aa->_ptr = aa->_base;
}

```

```

aa->_cnt = read(aa->_fd, aa->_ptr,
               aa->_flag & _UNBUF ? 1 : _BUFSIZE);
if (--aa->_cnt < 0) {
    if (aa->_cnt == -1)
        aa->_flag |= _EOF;
    else
        aa->_flag |= _ERR;
    aa->_cnt = 0;
    return(EOF);
}

return(*aa->_ptr++ & 0377); /* força caract. a ser > 0 */
}

```

A primeira chamada a `getc` para um arquivo particular encontra o contador em zero, o que força uma chamada a `_fillbuf`. Se `_fillbuf` vê que o arquivo não está aberto para leitura, ela retorna EOF imediatamente. Caso contrário, ela tenta alocar um buffer grande, e, falhando, um buffer para um único caractere, indicando o fato no sinalizador `_flag` de forma apropriada.

Uma vez que o buffer é estabelecido, `_fillbuf` simplesmente chama `read` para preenchê-lo, atualiza o contador e apontadores, e retorna o caractere do início do buffer. Chamadas subsequentes a `_fillbuf` encontrarão um buffer já alocado.

O que falta agora é saber como tudo se inicia. O arranjo `_iob` deve ser definido e inicializado para `stdin`, `stdout` e `stderr`:

```

FILE _iob[_NFILE] = {
    { NULL, 0, NULL, _READ, 0 }, /* stdin */
    { NULL, 0, NULL, _WRITE, 1 }, /* stdout */
    { NULL, 0, NULL, _WRITE | _UNBUF, 2 } /* stderr */
};

```

A inicialização dos sinalizadores da estrutura mostra `stdin` para leitura, `stdout` para gravação, e `stderr` para gravação sem buffer.

Exercício 8-3. Reescreva `fopen` e `_fillbuf` com campos ao invés de operações explícitas com bits.

Exercício 8-4. Projete e escreva as rotinas `_flushbuf` e `fclose`.

Exercício 8-5. A biblioteca padrão fornece a função

```
fseek(fp, deslocamento, origem)
```

que é idêntica a `lseek` exceto que `fp` é um apontador de arquivo ao invés de um descritor. Escreva `fseek`. Assegure que sua função `fseek` esteja coordenada com o tipo buffer (ou ausência de tal) usado pelas outras funções da biblioteca.

8.6 Exemplo – Listagem de Diretórios

Às vezes, um tipo diferente de interação com o sistema de arquivos é necessário: a determinação de informações sobre um arquivo e não sobre seu conteúdo. O comando `ls`

do UNIX ("list directory") é um exemplo — ele imprime os nomes dos arquivos de um diretório, e opcionalmente, outras informações, tais como tamanhos, permissões, e assim por diante.

Como, pelo menos no UNIX, um diretório é também um arquivo, não há nada de especial sobre um comando tal como o ls: ele lê um arquivo e obtém as partes relevantes de informação que ele encontra. No entanto, o formato da informação é determinado pelo sistema e não pelo usuário, de forma que ls precisa conhecer como o sistema representa as coisas.

Ilustraremos algumas dessas coisas escrevendo um programa chamado `tamarq` que é uma forma particular de ls, para imprimir o tamanho de todos os arquivos dados na sua lista de argumentos. Se um dos arquivos é um diretório, `tamarq` chama a si próprio de modo recursivo para processar este diretório. Na ausência de argumentos, ele processa o diretório corrente.

Para iniciar, uma breve revisão da estrutura do sistema de arquivos. Um diretório é um arquivo que contém uma lista de nomes de arquivos e alguma indicação de onde eles se localizam. A "localização" é, na realidade, um índice de outra tabela chamada "tabela de inodes". O inode para um arquivo é onde toda informação sobre o mesmo é guardada, com exceção de seu nome. Uma entrada no diretório consiste de apenas dois itens, um número de inode e o nome do arquivo. A especificação exata está contida no arquivo de inclusão `sys/dir.h` que contém:

```
#define    DIRSIZ    14    /* tamanho max do nome do arquivo */

struct direct    /* estrutura de uma entrada de um diretorio */
{
    ino_t d_ino;    /* numero do inode */
    char d_name[DIRSIZ];    /* nome do arquivo */
};
```

O "tipo" `ino_t` (definido como `typedef`) descreve o índice na tabela de inodes. Ele é `unsigned` no UNIX do PDP-11, mas poderia ser diferente em sistemas diferentes. Daí o `typedef`. Um conjunto completo dos tipos do "sistema" encontra-se em `sys/types.h`.

A função `stat` recebe o nome de um arquivo e retorna todas as informações do inode do arquivo (ou -1 se houver erro). Isto é,

```
struct stat bufst;
char *nome;

stat(nome, &bufst);
```

preenche a estrutura `bufst` com a informação do inode do arquivo indicado. A estrutura descrevendo o valor retornado por `stat` está em `sys/stat.h`:

```
struct stat    /* estrutura retornada por stat */
{
    dev_t    st_dev;    /* dispositivo do inode */
    ino_t    st_ino;    /* numero do inode */
    short    st_mode;    /* bits de modo */
    short    st_nlink;    /* numero de ligacoes do arquivo */
    short    st_uid;    /* identificação do dono */
    short    st_gid;    /* id. do grupo do dono */
};
```

```

dev_t    st_rdev;    /* para arquivos especiais */
off_t    st_size;    /* tamanho do arquivo em caracteres */
time_t    st_atime;    /* tempo do ultimo acesso */
time_t    st_mtime;    /* tempo da ultima modificacao */
time_t    st_ctime;    /* tempo de criacao */
};

```

A maioria dos campos está explicada pelos próprios comentários. O campo `st_mode` contém um conjunto de sinalizadores descrevendo o arquivo; por conveniência, as definições dos sinalizadores são também parte do arquivo `sys/stat.h`.

```

#define S_IFMT    0160000 /* tipo de arquivo */
#define S_IFDIR    0040000 /* diretorio */
#define S_IFCHR    0020000 /* especial do tipo caractere */
#define S_IFBLK    0060000 /* especial do tipo bloco */
#define S_IFREG    0100000 /* normal */
#define S_ISUID    04000 /* muda userid na execucao */
#define S_ISGID    02000 /* muda id do grupo na execucao */
#define S_ISVTX    01000 /* guarda texto permutado apos usar */
#define S_IREAD    0400 /* permissao de leitura */
#define S_IWRITE    0200 /* permissao de gravacao */
#define S_IXEXEC    0100 /* permissao de execucao */

```

Estamos prontos para escrever o programa `tamarq`. Se o modo obtido por `stat` indica que o arquivo não é um diretório, seu tamanho está disponível para ser impresso. Se ele é um diretório, entretanto, temos de processá-lo um arquivo por vez; ele por sua vez, pode conter subdiretórios, de forma que o processo é recursivo.

A rotina principal, como sempre, manipula principalmente os argumentos do comando; ela fornece cada argumento para a função `tamarq` num grande buffer.

```

#include <stdio.h>
#include <sys/types.h> /* typedefs */
#include <sys/dir.h> /* estrutura da entrada de diretorio */
#include <sys/stat.h> /* estrutura retornada por stat */
#define TAMBUF 256

main(argc, argv) /* tamarq: imprime tamanhos de arquivos */
char *argv[];
{
    char buf[TAMBUF];

    if (argc == 1) { /* default: diretorio corrente */
        strcpy(buf, ".");
        tamarq(buf);
    } else
        while (--argc > 0) {
            strcpy(buf, *++argv);
            tamarq(buf);
        }
}

```

A função `tamarq` imprime o tamanho do arquivo. Se o arquivo é um diretório, entretanto, `tamarq` chama primeiro `diretorio` para manipular os arquivos nele contidos. Observe o uso dos nomes dos sinalizadores `S_IFMT` e `S_IFDIR` de `stat.h`.

```
tamarq(nome)      /* imprime o tamanho do arquivo nome */
char *nome;
{
    struct stat bufst;

    if (stat(nome, &bufst) == -1) {
        fprintf(stderr, "tamarq: nao pode achar %s\n", nome);
        return;
    }
    if ((bufst.st_mode & S_IFMT) == S_IFDIR)
        diretorio(nome);
    printf("%8ld %s\n", bufst.st_size, nome);
}
```

A função `diretorio` é a mais complicada. Uma grande parte da rotina, entretanto, se relaciona com a criação do nome de percurso do arquivo a ser processado.

```
diretorio(nome)   /* tamarq para todos os arquivos em nome */
char *nome;
{
    struct direct bufdir;
    char *ain, *afn;
    int i, da;

    ain = nome + strlen(nome);
    *ain++ = '/'; /* adicionar barra ao nome do diretório */
    if (ain + DIRSIZ + 2 >= nome + BUFSIZE) /* nome muito grande */
        return;
    if ((da = open(nome, 0)) == -1)
        return;
    while (read(da, (char *)&bufdir, sizeof(bufdir)) > 0) {
        if (bufdir.d_ino == 0) /* entrada não usada */
            continue;
        if (strcmp(bufdir.d_name, ".") == 0
            || strcmp(bufdir.d_name, "..") == 0)
            continue; /* pula pai e a si mesmo */
        for (i = 0, afn = ain; i < DIRSIZ; i++)
            *afn++ = bufdir.d_name[i];
        *afn++ = '\0';
        tamarq(nome);
    }
    close(da);
    *--ain = '\0'; /* volta ao nome original */
}
```


Se uma entrada de diretório está desocupada (porque um arquivo foi removido), o índice do inode é zero, e essa posição é descartada. Cada diretório contém também entradas para si próprio, chamada ".", e para seu pai, ".."; evidentemente estas também devem ser descartadas, ou o programa executará indefinidamente.

Embora o programa *tamarq* seja especializado, ele indica algumas idéias importantes. Primeiro, muitos programas não são "software básico"; eles meramente usam informações cuja forma ou conteúdo é mantida pelo sistema operacional. Segundo, para tais programas, é crucial que a representação da informação apareça somente em arquivos padrões tais como *stat.h* e *dir.h* e que os arquivos sejam incluídos nos programas ao invés de embutir as declarações no programa.

8.7 Exemplo – Um Alocador de Memória

No Capítulo 5, apresentamos uma versão simples de *aloca*. A versão que escreveremos agora não tem restrição: chamadas a *aloca* e *libera* podem ser intercaladas em qualquer ordem; *aloca* chama o sistema operacional para obter mais memória quando for necessário. Além de sua utilidade direta, essas rotinas ilustram algumas considerações envolvendo a escrita de código dependente de máquina, de uma forma relativamente independente de máquina, e mostra também uma aplicação real de estruturas, uniões e *typedef*.

Ao invés de alocar espaço a partir de um arranjo com tamanho fixado em tempo de compilação, *aloca* requisitará espaço ao sistema operacional quando for necessário. Desde que outras atividades no programa podem também pedir espaço de forma assíncrona, o espaço que *aloca* gerencia pode não ser contíguo. Assim as áreas livres de armazenamento são guardadas numa cadeia de blocos livres. Cada bloco contém um tamanho, um apontador para o próximo bloco, e o espaço propriamente dito. Os blocos são guardados em ordem crescente de endereço, e o último bloco (maior endereço) aponta para o primeiro, de modo que a cadeia é na realidade, um anel.

Quando um pedido é feito, a lista livre é pesquisada até que um bloco de tamanho suficiente seja encontrado. Se o bloco tem o tamanho exato do pedido, ele é desligado da lista e retornado para o usuário. Se o bloco é maior, ele é dividido, e a quantidade apropriada é retornada para o usuário enquanto o resto é devolvido à lista livre. Se nenhum bloco com tamanho suficiente é encontrado, um outro bloco é obtido do sistema operacional e ligado à lista livre; então a pesquisa recomeça.

A liberação de espaço também causa uma pesquisa na lista livre, para encontrar o local apropriado para inserir o bloco liberado. Se o bloco sendo liberado é adjacente a um bloco da lista, ele é unido a este num bloco maior, para que o espaço não se torne muito fragmentado. A determinação da adjacência é fácil porque a lista livre é mantida em ordem de endereço.

Um problema, que avistamos no Capítulo 5, é o de garantir que o espaço retornado por *aloca* esteja alinhado apropriadamente para os objetos que serão armazenados nele. Embora as máquinas variem, para cada uma existe um tipo mais restritivo: se esse tipo mais restritivo pode ser armazenado num endereço particular, todos os outros tipos também podem sê-lo. Por exemplo, no IBM 360/370, o Honeywell 6000, e muitas outras máquinas, qualquer objeto pode ser armazenado num limite apropriado para um *double*; no PDP-11, *int* é suficiente.

Um bloco livre contém um apontador para o próximo bloco da cadeia, um registro do tamanho do bloco, e o espaço livre propriamente dito; a informação inicial de controle é chamada de "cabeçalho". Para simplificar o alinhamento, todos os tamanhos dos blocos

são múltiplos do tamanho do cabeçalho, e o cabeçalho é alinhado apropriadamente. Isto é obtido com uma união que contém o cabeçalho desejado e uma ocorrência do tipo mais restritivo:

```
typedef int ALINHA;      /* força alinhamento no PDP-11 */

union cabecalho {       /* cabeçalho de um bloco livre */
    struct {
        union cabecalho *apont; /* proximo bloco livre */
        unsigned tamanho; /* tamanho deste bloco livre */
    } s;
    ALINHA x;            /* força o alinhamento de blocos */
};

typedef union cabecalho CABECALHO;
```

Em `aloca`, o tamanho pedido em caracteres é arredondado para cima para unidades do tamanho do cabeçalho; o tamanho do bloco que será alocado é de uma unidade a mais que o calculado, para conter o cabeçalho, e este é o valor registrado no campo `tamanho` do cabeçalho. O apontador retornado por `aloca` aponta para o espaço livre, e não para o cabeçalho.

```
static CABECALHO base; /* lista vazia para começar */
static CABECALHO *aalloc = NULL; /* ultimo bloco alocado */

char *aloca (nbytes) /* alocador geral de memoria */
unsigned nbytes;
{
    CABECALHO *maismem();
    register CABECALHO *p, *q;
    register int unidades;

    unidades = 1 + (nbytes + sizeof(CABECALHO) - 1) / sizeof(CABECALHO);
    if ((q = aalloc) == NULL) { /* nao tem lista ainda */
        base.s.apont = aalloc = q = &base;
        base.s.tamanho = 0;
    }
    for (p = q->s.apont; q = p, p = p->s.apont) {
        if (p->s.tamanho >= unidades) { /* cabe */
            if (p->s.tamanho == unidades) /* exatamente */
                q->s.apont = p->s.apont;
            else { /* aloca no fim */
                p->s.tamanho += unidades;
                p += p->s.tamanho;
                p->s.tamanho = unidades;
            }
            aalloc = q;
            return((char *) (p + 1));
        }
    }
    if (p == aalloc) /* voltou ao inicio da lista */
```

```

        if ((p = maismem(nunidades)) == NULL)
            return(NULL);    /* nao tem mais */
    }
}

```

A variável *base* é usada para iniciar o processo; se *aloca* é *NULL*, por exemplo, na primeira chamada a *aloca*, uma lista livre degenerada é criada: ela contém um bloco de tamanho zero, e aponta para si própria. Em qualquer caso, a lista livre é então pesquisada. A pesquisa por um bloco livre de tamanho adequado começa no ponto (*aaloc*) onde o último bloco foi encontrado; esta estratégia ajuda a manter a lista homogênea. Se um bloco muito grande é encontrado, sua parte final é retornada para o usuário; desta forma apenas o campo de tamanho no cabeçalho original precisa ser ajustado. Em todos os casos, o apontador retornado para o usuário aponta para a área livre que está uma unidade além do cabeçalho. Observe que *p* é convertido para um apontador de caractere antes de ser retornado por *aloca*.

A função *maismem* obtém espaço do sistema operacional. Os detalhes de como isso é feito variam evidentemente de sistema para sistema. No UNIX, a chamada de sistema *sbrk(n)* retorna um apontador para *n* bytes de armazenamento (o apontador satisfaz todas as restrições de alinhamento). Visto que um pedido de memória para o sistema é uma operação comparativamente dispendiosa, não desejamos fazê-la em toda chamada a *aloca*, e, portanto, *maismem* arredonda o número de unidades requisitadas para um valor maior; este bloco maior será quebrado conforme as necessidades. A escala básica a ser usada é um parâmetro que pode ser mudado de acordo com a necessidade.

```

#define  NALOCA    128    /* num. de unidades alocadas de uma vez */

static CABECALHO *maismem(nu)    /* pede mais memoria ao sistema */
unsigned nu;
{
    char *sbrk( );
    register char *ac;
    register CABECALHO *au;
    register int rnu;

    rnu = NALOCA * ((nu + NALOCA - 1) / NALOCA);
    ac = sbrk(rnu * sizeof(CABECALHO));
    if ((int) ac == -1)    /* nao tem mais espaco */
        return(NULL);
    au = (CABECALHO *) ac;
    au->s.tamanho = rnu;
    libera((char *) (au + 1));
    return(aaloc);
}

```

sbrk retorna valor *-1* se não há espaço, muito embora *NULL* tivesse sido uma escolha melhor. O valor *-1* deve ser convertido para um *int* para que possa ser comparado corretamente. Novamente, moldes são muito usados para que a função seja relativamente imune a detalhes da representação de apontadores em máquinas diferentes.

libera é a última parte. Ela simplesmente pesquisa a lista livre, partindo de *aaloc*, procurando o local para inserir o bloco livre. Isto acontecerá entre dois blocos existentes

ou em uma das extremidades da lista. De qualquer maneira, se o bloco a ser liberado é adjacente a um ou a ambos os seus vizinhos, os blocos adjacentes são combinados. O único problema, é o de manter os apontadores apontando para as coisas certas e manter os tamanhos corretos.

```
libera(aa)      /* coloca o bloco aa na lista livre */
char *aa;
{
    register CABECALHO *p, *q;

    p = (CABECALHO *) aa - 1; /* aponta para o cabeçalho */
    for (q=aaloc; !(p > q && p < q->s.apont); q=q->s.apont)
        if (q >= q->s.apont && (p > q || p < q->s.apont))
            break; /* de um lado ou de outro */

    if (p + p->s.tamanho == q->s.apont) { /* junta ao de cima */
        p->s.tamanho += q->s.apont->s.tamanho;
        p->s.apont = q->s.apont->s.apont;
    } else
        p->s.apont = q->s.apont;
    if (q + q->s.tamanho == p) { /* junta ao de baixo */
        q->s.tamanho += p->s.tamanho;
        q->s.apont = p->s.apont;
    } else
        q->s.apont = p;
    aaloc = q;
}
```

Embora a alocação de memória seja intrinsecamente dependente de máquina, o código mostrado acima ilustra como as dependências de máquina podem ser confinadas e controladas numa parte muito pequena do programa. O uso de typedef e union controla o alinhamento (dado que `sbrk` fornece um apontador apropriado). Moldes explicitam as conversões de apontadores e até resolvem o problema de uma interface de sistema mal projetada. Embora os detalhes aqui relatados sejam relacionados com a alocação de memória, o enfoque geral é aplicável a outras situações também.

Exercício 8-6. A função `calloc(n, tam)` da biblioteca padrão retorna um apontador para `n` objetos de tamanho `tam`, inicializados com zero. Escreva `calloc` usando `aloca` como modelo ou como uma função a ser chamada.

Exercício 8-7. `aloca` aceita um pedido sem verificar a validade do tamanho pedido; libera acredita que o bloco que vai liberar tem um campo de tamanho válido. Melhore estas rotinas para que façam uma verificação de erros mais completa.

Exercícios 8-8. Escreva uma rotina `blibera(p, n)` que libere um bloco arbitrário `p` de `n` caracteres para a lista livre mantida por `aloca` e `libera`. Usando `blibera`, um usuário pode acrescentar um arranjo estático ou externo à lista livre em qualquer instante.

Apêndice A

MANUAL DE REFERÊNCIA C

1. Introdução

Este manual descreve a linguagem C nos computadores DEC PDP-11, Honeywell 6000, IBM System/370, e Interdata 8/32. Onde existem diferenças, concentra-se no PDP-11, mas tenta-se apontar os detalhes dependentes de implementação. Com raras exceções, estas dependências seguem diretamente das propriedades intrínsecas do hardware; os vários compiladores são muito compatíveis geralmente.

2. Convenções Léxicas

Há seis classes de símbolos: identificadores, palavras-chaves, constantes, cadeias, operadores e outros separadores. Brancos, caracteres de tabulação e de nova-linha e comentários (coletivamente chamados "espaços em branco") como descritos abaixo são ignorados exceto quando servem para separar símbolos. Algum espaço em branco é necessário para separar identificadores adjacentes, palavras-chaves e constantes.

Se a sequência de entrada for separada em símbolos até um dado caractere, o próximo símbolo é tomado como a maior cadeia de caracteres que poderia possivelmente constituir-se num símbolo.

2.1 Comentários

Os caracteres /* introduzem um comentário, que termina com os caracteres */. Comentários não podem ser aninhados.

2.2 Identificadores (Nomes)

Um identificador é uma sequência de letras e dígitos; o primeiro caractere deve ser uma letra. O sublinha conta como uma letra. Letras maiúsculas e minúsculas são diferentes. Os primeiros oito caracteres são significativos, embora mais possam ser usados. Identificadores externos, que são usados por vários montadores e carregadores, são mais restritos:

DEC PDP-11	7 letras, minúsculas diferentes de maiúsculas
Honeywell 6000	6 letras, minúsculas iguais a maiúsculas
IBM 360/370	7 letras, minúsculas iguais a maiúsculas
Interdata 8/32	8 letras, minúsculas diferentes de maiúsculas

2.3 Palavras-Chaves

Os seguintes identificadores são reservados para uso como palavras-chaves, e não podem ser usados de outra forma:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

A palavra-chave `entry` não está implementada atualmente por nenhum compilador mas é reservada para uso futuro. Algumas implementações reservam, também, as palavras `fortran` e `asm`.

2.4 Constantes

Há vários tipos de constantes, listados abaixo. Características de hardware que afetam tamanhos são identificados em 2.6.

2.4.1 Constantes inteiras

Uma constante inteira consistindo de uma sequência de dígitos é considerada octal se começa com 0 (dígito zero), e decimal caso contrário. Os dígitos 8 e 9 têm valor octal 10 e 11, respectivamente. Uma sequência de dígitos precedida por 0x ou 0X (dígito zero) é considerada hexadecimal inteira. Uma constante decimal cujo valor excede o maior inteiro com sinal da máquina é considerada long; uma constante octal ou hexadecimal que exceda o maior inteiro sem sinal da máquina é considerada long.

2.4.2 Constantes longas explícitas

Uma constante decimal, octal, ou hexadecimal imediatamente seguida por l (letra ele) ou L é considerada long. Como discutido abaixo, em algumas máquinas valores inteiros simples e longos podem ser considerados idênticos.

2.4.3 Constantes do tipo caractere

Uma constante do tipo caractere é um caractere entre apóstrofes, como em 'x'. O valor de uma constante do tipo caractere é o valor numérico do código do caractere no conjunto de caracteres da máquina.

Alguns caracteres não visíveis, o apóstrofo, e a contrabarra podem ser representados de acordo com a seguinte tabela de sequência de escape:

nova-linha	NL(LF)	\n
tabulação horizontal	HT	\t
retrocesso	BS	\b
retorno de carro	CR	\r
alimentação de formulário	FF	\f

contrabarra	\	\\
apóstrofo	'	'\
padrão de bits	ddd	\\ddd

A sequência de escape `\\ddd` consiste da contrabarra seguida por 1, 2 ou 3 dígitos octais que especificam o valor do caractere desejado. Um caso especial desta construção é `\\0` (não seguido por um dígito), que indica o caractere NUL. Se o caractere seguindo a contrabarra não é um dos especificados, a contrabarra é ignorada.

2.4.4 Constantes do tipo ponto flutuante

Uma constante do tipo ponto flutuante consiste de uma parte inteira, um ponto decimal, uma parte fracionária, um e ou E, e um expoente inteiro com sinal opcional. As partes inteira e fracionária consistem de seqüências de dígitos. Tanto a parte inteira ou fracionária (mas não ambas) podem ser omitidas; tanto o ponto decimal ou o e e o expoente (mas não ambos) podem ser omitidos. Toda constante do tipo ponto flutuante é considerada como sendo de dupla precisão.

2.5 Cadeias

Uma cadeia é uma seqüência de caracteres entre aspas, como em "...". Uma cadeia tem o tipo "arranjo de caracteres" e tem classe de armazenamento static (veja 4, abaixo) e é inicializada com os caracteres dados. Todas as cadeias, mesmo quando escritas de forma idêntica, são distintas (isto é, ocupam áreas diferentes de armazenamento). O compilador coloca um byte nulo `\\0` no final de cada cadeia para que os programas que pesquisam a cadeia possam encontrar seu fim. Em uma cadeia, a aspa deve ser precedida por uma contrabarra `\\`; além disso, as mesmas seqüências de escape descritas para constantes do tipo caractere podem ser usadas. Finalmente, uma contrabarra `\\` e uma nova-linha seguinte são ignoradas.

2.6 Características de Hardware

A tabela seguinte sumariza certas propriedades de hardware que variam de máquina para máquina. Embora isso afete a transportabilidade de programas, na prática não é um problema tão sério quanto se acharia *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64
faixa	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

Para essas quatro máquinas, números com ponto flutuante têm expoente de 8 bits.

3. Notação Sintática

Na notação sintática usada neste manual, categorias sintáticas são indicadas por impressão em *itálico*. Categorias alternativas são listadas em linhas separadas. Um símbolo terminal ou não-terminal opcional é indicado pelo caractere @; isto é

{ *expressão* @ }

indica uma expressão opcional entre chaves. A sintaxe é sumarizada em 18.

4. Em que Consiste um Nome?

C baseia a interpretação de um identificador em dois atributos do mesmo: sua *classe de armazenamento* e seu *tipo*. A classe de armazenamento determina a localização e o tempo de vida do armazenamento associado com o identificador; o tipo determina o significado dos valores encontrados na área de armazenamento do identificador.

Há quatro classes de armazenamento: automática, estática, externa e registrador. Variáveis automáticas são locais a cada invocação de um bloco (vide 9.2), e são descartadas no final do bloco; variáveis estáticas são locais a um bloco, mas retêm seu valores até a reentrada no bloco mesmo depois de ter sido deixado; variáveis externas existem e retêm seus valores durante a execução de todo o programa, e podem ser usadas para a comunicação entre funções, mesmo funções compiladas em separado. Variáveis do tipo registrador são (se possível) armazenadas em registradores rápidos da máquina; como variáveis automáticas, elas são locais a cada bloco e desaparecem no final do bloco.

C suporta vários tipos fundamentais de objetos:

Objetos declarados como caracteres (*char*) podem armazenar qualquer membro do conjunto de caracteres implementado, e se um caractere genuíno do conjunto de caracteres é armazenado numa variável do tipo caractere, seu valor é equivalente ao código inteiro do caractere. Outras quantidades podem ser armazenadas em variáveis do tipo caractere, mas a implementação é dependente de máquina.

Até três tamanhos de inteiros, declarados *short int*, *int*, e *long int* estão disponíveis. Inteiros longos não são menores que inteiros curtos, mas a implementação pode fazer um inteiro curto, ou um inteiro longo, ou ambos equivalentes a um inteiro normal. Inteiros normais têm o tamanho natural sugerido pela arquitetura da máquina hospedeira; os outros tamanhos existem para satisfazer necessidades especiais.

Inteiros sem sinal, declarados como *unsigned*, obedecem as leis da aritmética módulo 2^{**n} onde n é o número de bits na representação. (No PDP-11, quantidades longas sem sinal não são permitidas.)

Ponto flutuante de precisão simples (*float*) e de precisão dupla (*double*) podem ser sinônimos em algumas implementações.

Como os objetos precedentes podem ser interpretados como números, eles serão chamados tipos *aritméticos*. Os tipos *char* ou *int* de todos os tamanhos serão chamados tipos *inteiros*, *float* e *double* serão chamados tipos *flutuantes*.

Além dos tipos aritméticos fundamentais há uma classe conceitualmente infinita de tipos derivados construídos a partir dos tipos fundamentais das seguintes formas:

- arranjos* de objetos da maioria dos tipos;
- funções* que retornam objetos de um dado tipo;

apontadores para objetos de um dado tipo;
estruturas contendo uma seqüência de objetos de vários tipos;
uniões capazes de conter qualquer um de vários objetos de vários tipos.

Em geral, tais métodos de construções de objetos podem ser aplicados recursivamente.

5. Objetos e lvalues

Um *objeto* é uma região manipulável de armazenamento; um *lvalue* é uma expressão referindo-se a um objeto. Um exemplo óbvio de uma expressão lvalue é um identificador. Há operadores que produzem lvalues: por exemplo, se E é uma expressão do tipo apontador, então *E é uma expressão lvalue referindo-se ao objeto para o qual E aponta. O nome lvalue vem da expressão de atribuição E1 = E2 em que o operando esquerdo E1 deve ser uma expressão lvalue. A discussão de cada operador abaixo indica se ele espera operandos lvalue e se produz um lvalue.

6. Conversões

Um número de operadores pode, dependendo de seus operandos, causar uma conversão do valor de um operando de um tipo para outro. Essa seção explica os resultados a serem esperados de tais conversões. A subseção 6.6 sumariza as conversões executadas pela maioria dos operadores comuns; ele será complementado quando necessário na discussão de cada operador.

6.1 Caracteres e Inteiros

Um caractere ou um inteiro curto podem ser usados onde um inteiro o pode. Em todos os casos, o valor é convertido para um inteiro. A conversão de um inteiro curto para um longo sempre estende o sinal; inteiros são quantidades com sinal. A ocorrência ou não de extensão de sinal para caracteres depende da máquina, mas garante-se que um membro do conjunto de caracteres padrão nunca é negativo. Das máquinas consideradas neste manual, somente o PDP-11 estende o sinal. No PDP-11, variáveis do tipo caractere variam em valor de -128 a 127; os caracteres do alfabeto ASCII são todos positivos. Uma constante do tipo caractere especificada com uma seqüência de escape octal sofre uma extensão de sinal e pode ser negativa; por exemplo, '\377' tem valor -1.

Quando um inteiro longo é convertido para um inteiro mais curto ou para um char, ele é truncado à esquerda; os bits em excesso são simplesmente descartados.

6.2 Float e Double

Toda aritmética de ponto flutuante em C é executada em precisão dupla; sempre que um float aparece numa expressão ele é expandido para double com preenchimento de zeros da fração. Quando um double deve ser convertido para float, por exemplo numa atribuição, o double é arredondado antes de ser truncado para o tamanho de um float.

6.3 Ponto Flutuante e Integral

A conversão de valores de ponto flutuante para um tipo integral tende a ser dependente de máquina; em particular, a direção de truncamento de números negativos varia de máquina para máquina. O resultado é indefinido se o valor não cabe no espaço fornecido.

Conversões de valores integrais para o tipo ponto flutuante são bem comportadas. Alguma perda de precisão ocorre se o destino não dispõe de bits suficientes.

6.4 Apontadores e Inteiros

Um inteiro ou um inteiro longo podem ser adicionados ou subtraídos de um apontador; em tal caso, o primeiro é convertido como especificado na discussão do operador de adição.

Dois apontadores para objetos do mesmo tipo podem ser subtraídos; neste caso o resultado é convertido para um inteiro como especificado na discussão do operador de subtração.

6.5 Unsigned

Sempre que um inteiro sem sinal e um inteiro normal são combinados, o inteiro normal é convertido para unsigned e o resultado é unsigned. O valor é o menor inteiro sem sinal congruente com o inteiro normal (módulo 2 elevado ao tamanho da palavra). Na representação em complemento de 2, tal conversão é conceitual e não há mudança no padrão de bits.

Quando um inteiro sem sinal é convertido para long, o valor do resultado é numericamente igual ao do inteiro sem sinal. A conversão é feita com o preenchimento de zeros à esquerda.

6.6 Conversões Aritméticas

Muitos operadores causam conversões e produzem tipos resultantes de modo similar. Este padrão será chamado de “conversão aritmética normal”.

Primeiro, quaisquer operandos do tipo char ou short são convertidos para int, e quaisquer do tipo float são convertidos para double.

Então, se um dos dois operandos é double, o outro é convertido para double e esse é o tipo do resultado.

Senão, se um dos dois operandos é long, o outro é convertido para long e esse é o tipo do resultado.

Senão, se um dos dois operandos é unsigned, o outro operando é convertido para unsigned e esse é o tipo do resultado.

Senão, ambos os operandos devem ser int, e esse é o tipo do resultado.

7. Expressões

A precedência dos operadores de expressões segue a ordem das subseções principais dessa seção, maior precedência primeiro. Então, por exemplo, as expressões referenciadas como sendo operandos de + (subseção 7.4) são aquelas definidas nas subseções 7.1-7.3. Em cada subseção, os operadores têm a mesma precedência. A associatividade à esquerda ou à direita é especificada em cada subseção para os operadores nela discutidos. A precedência e associatividade de todos os operadores de expressão é resumida na gramática na seção 18.

Além dessas regras, a ordem de avaliação de expressões é indefinida. Em particular, o compilador calcula subexpressões na ordem que ele acredita ser a mais eficiente, mesmo se as subexpressões envolvem efeitos colaterais. A ordem em que os efeitos colaterais

ocorrem não é especificada. Expressões envolvendo um operador comutativo e associativo (*, +, &, |, ~) podem ser rearrumadas arbitrariamente, mesmo na presença de parênteses. Para forçar uma ordem particular de avaliação deve-se usar variáveis temporárias explícitas.

O tratamento de transbordo e a verificação de divisão na avaliação de expressões são dependentes de máquina. Todas as implementações existentes de C ignoram transbordos com inteiros; o tratamento de divisão por zero, e de todas as exceções envolvendo ponto flutuante, varia entre máquinas, e é normalmente modificável com uma função de biblioteca.

7.1 Expressões Primárias

Expressões primárias envolvendo ., ->, indexação e chamadas de funções se agrupam da esquerda para a direita.

expressao-primaria:
 identificador
 constante
 cadeia
 (*expressao*)
 expressao-primaria [*expressao*]
 expressao-primaria (*lista-expressao* @)
 lvalue-primario . *identificador*
 expressao-primaria -> *identificador*

lista-expressao:
 expressao
 lista-expressao, *expressao*

Um identificador é uma expressão primária, desde que ele tenha sido adequadamente declarado como discutido abaixo. Seu tipo é especificado por sua declaração. Se o tipo do identificador é "arranjo de . . .", entretanto, o valor da expressão correspondendo ao identificador é um apontador para o primeiro objeto do arranjo e o tipo da expressão é "apontador para . . .". Além do mais, um identificador de arranjo não é uma expressão lvalue. Da mesma forma, um identificador que é declarado como sendo uma "função retornando . . ." quando usado sem ser na posição do nome da função de uma chamada, é convertido para "apontador para função retornando . . .".

Uma constante é uma expressão primária. Seu tipo pôde ser int, long, ou double, dependendo de sua forma. Constantes do tipo caractere têm tipo int; constantes do tipo flutuante são double.

Uma cadeia é uma expressão primária. Seu tipo é originalmente "arranjo de char"; mas, seguindo a mesma regra dada acima para identificadores, isto é, modificado para "apontador para char" e o resultado é um apontador para o primeiro caractere da cadeia. (Há uma exceção em certos inicializadores; vide a subseção 8.6.)

Uma expressão parentetizada é uma expressão primária cujo tipo e valor são idênticos aos da expressão sem parênteses. A precedência dos parênteses não modifica o fato de uma expressão ser ou não um lvalue.

Uma expressão primária seguida por uma expressão entre colchetes é uma expressão primária. O significado intuitivo é o de indexação. Normalmente, a expressão primária tem o tipo "apontador para . . .", o índice é int, e o tipo do resultado é ". . .". A ex-

pressão $E1 [E2]$ é idêntica (por definição) a $*((E1) + (E2))$. Todas as dicas necessárias para entender essa notação estão contidas nessa seção juntamente com a discussão nas subseções 7.1, 7.2, e 7.4 sobre identificadores, *, e +, respectivamente; a subseção 14.3 resume as implicações.

Uma chamada de função é uma expressão primária seguida por parênteses contendo uma lista (possivelmente vazia) de expressões separadas por vírgula constituindo os argumentos atuais para a função. A expressão primária precisa ser do tipo "função retornando . . .", e o resultado da chamada da função é do tipo ". . .". Como indicado abaixo, um identificador não definido e imediatamente seguido por um parêntese esquerdo é contextualmente declarado como representando uma função que retorna um inteiro; desta forma, nos casos mais comuns, funções retornando um valor inteiro não precisam ser declaradas.

Qualquer argumento atual do tipo float é convertido para double antes da chamada; qualquer argumento do tipo char ou short é convertido para int; e, como sempre, nomes de arranjos são convertidos em apontadores. Nenhuma outra conversão é feita automaticamente; em particular, o compilador não compara os tipos dos argumentos atuais com os dos formais. Se uma conversão for necessária, use um molde; vide as subseções 7.2 e 8.7.

Na preparação da chamada a uma função, é feita uma cópia de cada parâmetro atual; assim, toda passagem de argumentos em C é feita por valor. Uma função pode alterar os valores dos seus parâmetros formais, mas tal mudança não afeta o valor dos parâmetros atuais. Por outro lado, é possível passar um apontador, entendendo-se que a função poderá mudar o valor do objeto apontado. Um nome de arranjo é uma expressão do tipo apontador. A ordem de avaliação dos argumentos é indefinida pela linguagem; observe que os vários compiladores diferem entre si.

Chamadas recursivas para qualquer função são permitidas.

Uma expressão primária seguida por um ponto, seguido por um identificador é uma expressão. A primeira expressão deve ser um lvalue indicando uma estrutura ou união, e o identificador deve ser o nome de um membro da estrutura ou união. O resultado é um lvalue referindo-se ao membro da estrutura ou união.

Uma expressão primária seguida por uma seta (construída com - e >) seguida por um identificador é uma expressão. A primeira expressão deve ser um apontador para uma estrutura ou união e o identificador deve indicar um membro da estrutura ou união. O resultado é um lvalue referindo-se ao membro identificado da estrutura ou união, apontada pela expressão.

Desta forma, a expressão $E1 -> MDE$ é idêntica a $(*E1).MDE$. Estruturas e uniões são discutidas na subseção 8.5. As regras dadas aqui para o uso de estruturas e uniões não são seguidas com todo rigor, de modo a permitir alguma fuga do mecanismo de tipos. Vide a subseção 14.1.

7.2 Operadores Unários

Expressões com operadores unários se associam da direita para a esquerda:

expressao-unaria:

- * *expressao*
- & *lvalue*
- *expressao*
- ! *expressao*

```

    ~ expressao
    ++ value
    -- lvalue
    lvalue ++
    lvalue --
    (nome-de-tipo) expressao
    sizeof expressao
    sizeof (nome-de-tipo)

```

O operador unário `*` significa *indireção*: a expressão deve ser um apontador, e o resultado é um `lvalue` referindo-se ao objeto apontado pela expressão. Se o tipo da expressão é “apontador para . . .”, o tipo do resultado é “. . .”.

O resultado do operador unário `&` é um apontador para o objeto referenciado pelo `lvalue`. Se o tipo do `lvalue` é “. . .”, o tipo do resultado é “apontador para . . .”.

O resultado do operador unário `-` é o negativo do seu operando. As conversões aritméticas normais são executadas. O negativo de uma quantidade sem sinal é calculada pela subtração da mesma de 2^n , onde n é o número de bits num int. Não há operador unário `+`.

O resultado do operador de negação lógica `!` é 1 se o valor do seu operando é 0, e 0 se o valor do seu operando é diferente de zero. O tipo do resultado é `int`. Ele é aplicado a qualquer tipo aritmético ou a apontadores.

O operador `~` produz o complemento de um de seu operando. As conversões aritméticas normais são executadas. O tipo do operando deve ser integral.

O objeto referenciado pelo operando `lvalue` do operador `++` prefixado é incrementado. O valor é o novo valor do operando, mas não é um `lvalue`. A expressão `++x` é equivalente a `x++ = 1`. Vide a discussão da adição (subseção 7.4) e da atribuição (subseção 7.14) para informação sobre conversões.

O operando `lvalue` do operador `--` prefixado é decrementado como no caso do `++` prefixado.

Quando o `++` pós-fixado é aplicado a um `lvalue`, o resultado é o valor do objeto referenciado pelo `lvalue`. Após o uso do resultado, o objeto é incrementado como no caso do operador `++` prefixado. O tipo do resultado é o tipo da expressão `lvalue`.

Quando o `--` pós-fixado é aplicado a um `lvalue`, o resultado é o valor do objeto referenciado por `lvalue`. Após o uso do resultado, o objeto é decrementado como no caso do operador `--` prefixado. O tipo do resultado é o tipo da expressão `lvalue`.

Uma expressão precedida por um nome de um tipo de dados entre parênteses provoca a conversão do valor da expressão para o tipo especificado. Tal construção é chamada “molde”. Nomes de tipos são descritos na subseção 8.7.

O operador `sizeof` produz o tamanho, em bytes, de seu operando. (Um `byte` é indefinido pela linguagem exceto em termos do valor de `sizeof`. Entretanto, em todas as versões existentes um `byte` é o espaço necessário para se armazenar um `char`.) Quando aplicado a um arranjo, o resultado é o número total de bytes no arranjo. O tamanho é determinado a partir da declaração dos objetos na expressão. Essa expressão é semanticamente uma constante inteira e pode ser usada em qualquer lugar onde uma constante é necessária. Seu uso maior é na comunicação com rotinas tais como alocadores de memória e sistemas de entrada e saída.

O operador `sizeof` pode ser aplicado também a um nome de tipo entre parênteses. Neste caso, ele produz o tamanho, em bytes, de um objeto do tipo indicado.

A construção `sizeof (tipo)` é considerada como uma unidade; por exemplo, a expressão: `sizeof (tipo) - 2` significa `{sizeof (tipo)} - 2`.

7.3 Operadores Multiplicadores

Os operadores multiplicadores `*`, `/`, e `%` são associativos da esquerda para a direita. As conversões aritméticas normais são executadas.

expressao-multiplicadora:

*expressao * expressao*

expressao / expressao

expressao % expressao

O operador binário `*` indica multiplicação. O operador `*` é associativo e expressões com várias multiplicações no mesmo nível podem ser rearrumadas pelo compilador.

O operador binário `/` indica divisão. Quando inteiros positivos são divididos, o truncamento tende a 0, mas a forma de truncamento é dependente de máquina se um dos operandos é negativo. Em todas as máquinas cobertas por esse manual, o resto tem o sinal do dividendo. A expressão $(a/b)*b + a\%b$ sempre é igual a a (se b é diferente de 0).

O operador binário `%` produz o resto da divisão da primeira expressão pela segunda. As conversões aritméticas normais são executadas. Os operandos não devem ser do tipo `float`.

7.4 Operadores Adicionadores

Os operadores adicionadores `+` e `-` são associativos da esquerda para a direita. As conversões aritméticas normais são executadas. Há algumas possibilidades adicionais de tipo para cada operador.

expressao-adicionadora:

expressao + expressao

expressao - expressao

O resultado do operador `+` é a soma dos seus operandos. Um apontador para um objeto em um arranjo e um valor de qualquer tipo integral podem ser adicionados. O valor integral é convertido, em todos os casos, num deslocamento de endereço pela multiplicação do mesmo pelo tamanho do objeto para o qual o apontador aponta. O resultado é um apontador do tipo do apontador original, e aponta para outro objeto no mesmo arranjo, com deslocamento apropriado com relação ao objeto original. Assim, se p aponta para um objeto em um arranjo, a expressão $p + 1$ é um apontador para o próximo elemento do arranjo.

Nenhuma outra combinação de tipos é permitida para apontadores.

O operador `+` é associativo e expressões com várias somas podem ser rearrumadas pelo compilador.

O resultado do operador `-` é a diferença de seus operandos. As conversões aritméticas normais são executadas. Adicionalmente, um valor de qualquer tipo integral pode ser subtraído de um apontador, e as mesmas conversões vistas no caso de adição são aplicadas.

Se dois apontadores para objetos do mesmo tipo são subtraídos, o resultado é convertido (pela divisão pelo tamanho do objeto) em um `int` representando o número de objetos separando os apontadores. Tal conversão dará, em geral, resultados inesperados se os apontadores não apontarem para objetos do mesmo arranjo, já que apontadores, mesmo

para objetos do mesmo tipo, não diferem necessariamente de um múltiplo do tamanho do objeto.

7.5 Operadores de Deslocamento

Os operadores de deslocamento $<<$ e $>>$ são associativos da esquerda para a direita. Ambos executam as conversões aritméticas normais em seus operandos, cada um dos quais deve ser integral. Então o operando direito é convertido para int; o tipo do resultado é o do operando esquerdo. O resultado é indefinido se o operando direito é negativo, ou maior ou igual ao tamanho do objeto em bits.

expressao-deslocamento:

expressao $<<$ *expressao*

expressao $>>$ *expressao*

O valor de $E1 << E2$ é $E1$ (interpretada como um padrão de bits) deslocada à esquerda de $E2$ bits; os bits vagos são preenchidos com zero. O valor de $E1 >> E2$ é $E1$ deslocada à direita de $E2$ bits. O deslocamento à direita é lógico (preenchimento com zero) se $E1$ é unsigned; caso contrário, ele pode ser (como no caso do PDP-11) aritmético (preenchimento com o bit de sinal).

7.6 Operadores Relacionais

Os operadores relacionais são associativos da esquerda para a direita, mas esse fato não é muito útil, $a < b < c$ não significa o que aparenta ser.

expressao-relacional:

expressao $<$ *expressao*

expressao $>$ *expressao*

expressao $<=$ *expressao*

expressao $>=$ *expressao*

Os operadores $<$ (menor que), $>$ (maior que), $<=$ (menor ou igual a) e $>=$ (maior ou igual a) produzem 0 se a relação especificada é falsa e 1 se é verdadeira. O tipo do resultado é int. As conversões aritméticas normais são executadas. Dois apontadores podem ser comparados; o resultado depende das localizações relativas no espaço de endereçamento dos objetos apontados. A comparação de apontadores é transportável somente quando apontadores apontam para objetos do mesmo arranjo.

7.7 Operadores de Igualdade

expressao-de-igualdade:

expressao $==$ *expressao*

expressao $!=$ *expressao*

Os operadores $==$ (igual a) e $!=$ (não igual a) são exatamente análogos aos operadores relacionais exceto por sua menor precedência. Então $a < b == c < d$ é 1 sempre que $a < b$ e $c < d$ tenham o mesmo valor booleano.

Um apontador pode ser comparado com um inteiro, mas o resultado é dependente de máquina, a menos que o inteiro seja a constante 0. Um apontador para o qual foi atribuído o valor 0 não aponta para nenhum objeto, e aparece como sendo igual a 0; por convenção, tal apontador é considerado nulo.

7.8 Operador e Bit-a-bit

expressao-e:
expressao & expressao

O operando `&` é associativo e expressões envolvendo o mesmo podem ser rearrumadas. As conversões aritméticas normais são executadas; o resultado é o produto lógico dos operandos. O operador aplica-se somente a operandos integrais.

7.9 Operador ou-exclusivo Bit-a-bit

expressao-ou-exclusivo:
expressao ^ expressao

O operador `^` é associativo e expressões envolvendo o mesmo podem ser rearrumadas. As conversões aritméticas normais são executadas; o resultado é a soma lógica exclusiva dos operandos. O operador aplica-se somente a operandos integrais.

7.10 Operador ou-inclusivo Bit-a-bit

expressao-ou-inclusivo:
expressao | expressao

O operador `|` é associativo e expressões envolvendo o mesmo podem ser rearrumadas. As conversões aritméticas normais são executadas; o resultado é a soma lógica inclusiva dos operandos. O operador aplica-se somente a operandos integrais.

7.11 Operador e Lógico

expressao-e-logico:
expressao && expressao

O operador `&&` é associativo da esquerda para a direita. Ele retorna 1 se ambos seus operandos são diferentes de zero, 0 caso contrário. Diferentemente de `&`, `&&` garante a avaliação da esquerda para a direita; além do mais, o segundo operando não é avaliado se o primeiro operando é 0.

Os operandos não precisam ter o mesmo tipo, mas cada um deve ter um dos tipos fundamentais ou ser um apontador. O resultado é sempre do tipo `int`.

7.12 Operador ou Lógico

expressao-ou-logico:
expressao || expressao

O operador `||` é associativo da esquerda para a direita. Ele retorna 1 se um dos operandos é diferente de zero, 0 caso contrário. Diferentemente de `|`, `||` garante a avaliação da esquerda para a direita; além do mais o segundo operando não é avaliado se o valor do primeiro é diferente de zero.

Os operandos não precisam ter o mesmo tipo, mas cada um deve ter um dos tipos fundamentais ou ser um apontador. O resultado é sempre do tipo `int`.

7.13 Operador Condicional

expressao-condicional:
expressao ? expressao : expressao

Expressões condicionais são associativas da direita para a esquerda. A primeira expressão é avaliada e, se é diferente de zero, o resultado é o valor da segunda expressão, caso contrário é o valor da terceira. Se possível, as conversões aritméticas normais são executadas para tornar a segunda e a terceira expressões do mesmo tipo; caso contrário, se ambas são apontadores do mesmo tipo, o resultado tem este tipo, caso contrário, uma deve ser um apontador e a outra a constante 0, e o resultado tem o tipo do apontador. Somente uma, entre a segunda e a terceira expressão, é avaliada.

7.14 Operadores de Atribuição

Há um número de operadores de atribuição, todos associativos da direita para a esquerda. Todos requerem um lvalue como seu operando esquerdo, e o tipo de uma expressão de atribuição é o do seu operando esquerdo. O valor é o armazenado no operando esquerdo após a atribuição. As duas partes de um operador de atribuição composto são símbolos independentes.

expressao-de-atribuição:

lvalue = expressao

lvalue += expressao

lvalue -= expressao

*lvalue *= expressao*

lvalue /= expressao

lvalue %= expressao

lvalue >> = expressao

lvalue << = expressao

lvalue & = expressao

lvalue ^ = expressao

lvalue |= expressao

Na atribuição simples com $=$, o valor da expressão substitui o valor do objeto referenciado pelo lvalue. Se ambos os operandos são de tipo aritmético, o operando direito é convertido para o tipo do operando esquerdo antes da atribuição.

O comportamento de uma expressão da forma $E1\ op\ =\ E2$ pode ser deduzido pela observação da expressão equivalente $E1\ =\ E1\ op\ (E2)$; entretanto, $E1$ é avaliada somente uma vez. Em $+=$ e $-=$, o operando esquerdo pode ser um apontador, e, neste caso, o operando direito (integral) é convertido como explicado na subseção 7.4. Todo operando direito e operando esquerdo não apontador devem ser de tipo aritmético.

Os compiladores atuais permitem a atribuição de um apontador a um inteiro, um inteiro a um apontador, e um apontador para um apontador de outro tipo. A atribuição é uma simples operação de cópia, sem conversão. Tal uso não é transportável, e pode produzir apontadores que provocam exceções de endereçamento quando usados. Entretanto, é garantido que a atribuição da constante 0 a um apontador produzirá um apontador nulo distinto de um apontador para qualquer objeto.

7.15 Operador Vírgula

expressao-virgula:

expressao , expressao

Um par de expressões separadas por uma vírgula é avaliado da esquerda para a direita e o valor da expressão à esquerda é descartado. O tipo e o valor do resultado são o tipo e o

valor do operando direito. Esse operador é associativo da esquerda para a direita. Em contextos onde a vírgula tem um significado especial, por exemplo numa lista de argumentos atuais para funções (subseção 7.1) e listas de inicializadores (subseção 8.6), o operador vírgula descrito aqui só pode aparecer entre parênteses; por exemplo

f (a, (t = 3, t + 2), c)

tem três argumentos, o segundo dos quais tem valor 5.

8. Declarações

Declarações são usadas para especificar a interpretação dada por C a cada identificador; elas não necessariamente reservam armazenamento para o identificador. Declarações têm a forma:

declaração:

espec-de-decl lista-de-declaradores @;

Os declaradores na lista-de-declaradores contêm os identificadores sendo declarados. Os *espec-de-decl* consistem de uma sequência de especificadores de classe de armazenamento e de tipo.

espec-de-decl:

espec-tipo *espec-de-decl* @

espec-ca *espec-de-decl* @

A lista deve ser consistente na forma descrita abaixo.

8.1 Especificadores de Classe de Armazenamento

Os *espec-ca* são

espec-ca:

auto

static

extern

register

typedef

O especificador *typedef* não reserva armazenamento e é chamado de “especificador de classe de armazenamento” somente por conveniência sintática; ele é discutido na subseção 8.8. Os significados das várias classes de armazenamento foram discutidos na seção 4.

As declarações *auto*, *static*, e *register* também servem como definições no sentido de que elas causam a reserva de uma quantidade apropriada de armazenamento. No caso *extern* deve haver uma definição externa (seção 10) para os identificadores externos em algum lugar fora da função em que eles são declarados.

Uma declaração *register* pode ser vista como uma declaração *auto*, junto com a indicação para o compilador de que as variáveis serão frequentemente utilizadas. Somente as primeiras poucas declarações são efetivas. Além do mais, somente variáveis de certos tipos serão armazenadas em registradores; no PDP-11, elas são *int*, *char*, ou *apontador*. Uma outra restrição se aplica a variáveis do tipo *registrador*: o operador *&* (endereço de) não pode ser aplicado a eles. Programas menores e mais rápidos são obtidos se declarações de registradores forem usadas apropriadamente, mas a otimização de código futura pode torná-las desnecessárias.

No máximo um *espec-ca* pode ser dado numa declaração. Se for omitido numa declaração, é assumido auto dentro de uma função e extern fora. Exceção: funções nunca são automáticas.

8.2 Especificadores de Tipo

Os *espec-tipo* são:

espec-tipo:
char
short
int
long
unsigned
float
double
espec-de-estrutura-ou-uniao
nome-typedef

As palavras *long*, *short* e *unsigned* podem ser vistas como adjetivos; as seguintes combinações são aceitáveis:

short int
long int
unsigned int
long float

O significado da última construção é *double*. No máximo um especificador de tipo pode ser dado numa declaração. Se for omitido numa declaração, é assumido ser *int*.

Especificadores para estruturas e uniões são discutidos na subsecção 8.5; declarações com nomes de *typedef* são discutidas na subsecção 8.8.

8.3 Declaradores

A *lista-de-declaradores* que aparece numa declaração é uma sequência de declaradores separados por vírgula, cada um dos quais podendo ter um inicializador.

lista-de-declaradores:
declarador-inic
declarador-inic , *lista-de-declaradores*
declarador-inic:
declarador inicializador @

Inicializadores são discutidos na subsecção 8.6. Os especificadores na declaração indicam a classe de armazenamento e o tipo dos objetos aos quais os declaradores se referem. Declaradores têm a seguinte sintaxe:

declarador:
identificador
(declarador)
* *declarador*
declarador ()
declarador [expressao-constante @]

O agrupamento é igual ao caso de expressões.

8.4 Significado de Declaradores

Cada declarador é considerado como uma declaração de que quando uma construção da mesma forma do declarador aparecer em uma expressão, produzirá um objeto de classe de armazenamento e tipo indicados. Cada declarador contém exatamente um identificador; ele é o identificador que é declarado.

Se um identificador sem parênteses aparece como um declarador, ele tem classe de armazenamento e tipo indicados pelo especificador no início da declaração.

Um declarador entre parênteses é idêntico ao sem parênteses, mas a ligação de declaradores complexos pode ser alterada com parênteses. Veja os exemplos abaixo.

Agora imagine a declaração

T D1

onde T é um espec-tipo (tal como int, etc.) e D1 é um declarador. Suponha que esta declaração faça com que o identificador tenha tipo "... T", onde "..." é vazio se D1 é um identificador simples (assim o tipo de x em int x" é int). Então se D1 tem a forma

*D

o tipo do identificador é "... apontador para T". Se D1 tem a forma

D ()

o tipo do identificador é "... função retornando T". Se D1 tem a forma

D [expressão-constante]

ou

D []

então o tipo do identificador é "... arranjo de T". No primeiro caso a expressão constante é uma expressão cujo valor é determinável quando da compilação, e cujo tipo é int (expressões constantes são definidas com precisão na seção 15). Quando várias especificações de "arranjo de" são adjacentes, um arranjo multidimensional é criado; as expressões constantes que especificam os limites dos arranjos podem ser omitidas somente para o primeiro membro da sequência. Esta omissão é útil quando o arranjo é externo e a definição que aloca armazenamento é dada em outro local. A primeira expressão constante pode ser omitida também quando o declarador é seguido de inicialização. Neste caso, o tamanho é calculado a partir do número de elementos iniciais fornecidos.

Um arranjo pode ser construído a partir de um dos tipos básicos, de um apontador, de uma estrutura ou união, e de outro arranjo (para gerar um arranjo multidimensional).

Nem todas as possibilidades disponíveis pela sintaxe acima são permitidas atualmente. As restrições são as seguintes: funções não podem retornar arranjos, estruturas, uniões ou funções, embora elas possam retornar apontadores para tais coisas; não há arranjos de funções, embora possa haver arranjos de apontadores para funções. Da mesma forma, uma estrutura ou união não pode conter uma função, mas pode conter um apontador para uma função.

Como exemplo, a declaração

int i, *ai, f (), *fai (), (*afi) ();

declara um inteiro i, um apontador ai para um inteiro, uma função f que retorna um inteiro, uma função fai que retorna um apontador para um inteiro, e um apontador afi para uma função que retorna um inteiro. Compare os dois últimos. A ligação de *fai () é *{fai

()); a declaração sugere, e a mesma construção numa expressão requer, a chamada de uma função *fai* e então o uso de indireção usando o resultado (apontador) para produzir um inteiro. No declarador (**afi*) (), os parênteses adicionais são necessários, como eles o são numa expressão, para indicar que a indireção através de um apontador para uma função produz uma função, a qual é então chamada; ela retorna um inteiro.

Como um outro exemplo, temos:

```
float af [17], *aaf [17];
```

declara um arranjo de números float e um arranjo de apontadores para números float. Finalmente,

```
static int x3d [3] [5] [7];
```

declara um arranjo tridimensional estático de inteiros com dimensões 3x5x7. Em mais detalhes, *x3d* é um arranjo de três itens; cada item é um arranjo de cinco arranjos; cada um dos últimos arranjos é um arranjo de sete inteiros. Qualquer uma das expressões *x3d*, *x3d* [*i*], *x3d* [*i*] [*j*], *x3d* [*i*] [*j*] [*k*] pode aparecer numa expressão. As três primeiras têm tipo “arranjo”, a última tem tipo *int*.

8.5 Declarações de Estrutura e União

Uma estrutura é um objeto consistindo de uma sequência de membros com nomes. Cada membro pode ser de qualquer tipo. Uma união é um objeto que pode, em um dado instante, conter qualquer um dentre vários membros. Especificadores de estrutura e união têm a mesma forma

espec-de-estr-ou-uniao:

estr-ou-uniao {*lista-de-decl-de-estr*}

estr-ou-uniao *identificador* {*lista-de-decl-de-estr*}

estr-ou-uniao *identificador*

estr-ou-uniao:

struct

union

A *lista-de-decl-de-estr* é uma sequência de declarações para os membros da estrutura ou união:

lista-de-decl-de-estr:

decl-estr

decl-estr *lista-de-decl-de-estr*

decl-estr:

espec-tipo *lista-declarador-estr*;

lista-declarador-estr:

declarador-estr

declarador-estr , *lista-declarador-estr*

No caso comum, um *declarador-estr* é apenas um declarador para um membro de uma estrutura ou união. Um membro de uma estrutura pode ser constituído também de um nú-

mero especificado de bits. Tal membro é chamado de *campo*; seu tamanho é separado do nome do campo por dois-pontos.

declarador-estr:

declarador

declarador : expressao-constante

: expressao-constante

Dentro de uma estrutura, os objetos declarados têm endereços crescentes de acordo com suas posições da esquerda para a direita. Cada membro que não é um campo começa num limite de endereço apropriado para seu tipo; assim, pode haver buracos sem nomes numa estrutura. Campos são compactados em inteiros da máquina; eles não atravessam palavras. Um campo que não cabe no espaço restante de uma palavra é colocado na próxima palavra. Nenhum campo pode ser maior que uma palavra. Campos são colocados da direita para a esquerda no PDP-11, e da esquerda para a direita em outras máquinas.

Um declarador-estr sem declarador, somente um dois-pontos e um tamanho, indica um campo sem nome, e é usado para preencher espaços requeridos por definições de formatos impostas externamente. Um caso especial é o campo sem nome com tamanho 0 que força o alinhamento do próximo campo no início da próxima palavra. O "próximo campo" presumivelmente é um campo, e não um membro comum da estrutura, porque nesse caso o alinhamento seria automático.

A linguagem não restringe os tipos das coisas que são declaradas como campos, mas as implementações não necessariamente devem aceitar outros tipos além de campo inteiros. Além do mais, mesmo campos do tipo int podem ser considerados como sendo unsigned. No PDP-11, campos não têm sinal e contêm somente valores inteiros. Em todas as implementações, não há arranjos de campos, e o operador & (endereço de) não pode ser aplicado a eles, e, portanto, não há apontadores para campos.

Uma união pode ser vista como uma estrutura cujos campos começam todos com deslocamento 0 e cujo tamanho é suficiente para conter qualquer um de seus membros. No máximo um dos membros pode ser armazenado na união em um dado instante.

Um especificador de estrutura ou união da segunda forma, isto é, um dos seguintes

struct identificador {lista-de-decl-de-estr}

union identificador {lista-de-decl-de-estr}

declara o identificador como sendo a *etiqueta da estrutura* (ou da união) especificada pela lista. Uma declaração subsequente pode então usar a terceira forma do especificador, isto é, uma das seguintes construções:

struct identificador

union identificador

Etiquetas de estruturas permitem a definição de estruturas auto-referenciáveis; elas permitem também que a declaração completa seja fornecida uma única vez e seja usada várias vezes. É ilegal declarar uma estrutura ou união que contém uma ocorrência de si própria, mas uma estrutura ou união pode conter um apontador para uma ocorrência de si própria.

Os nomes dos membros e etiquetas podem ser iguais a variáveis ordinárias. Entretanto, as etiquetas e membros devem ser mutuamente distintos.

Duas estruturas podem compartilhar uma sequência inicial de membros comuns, isto é, o mesmo membro pode aparecer em duas estruturas diferentes se tem o mesmo tipo

em ambas e se todos os membros anteriores são iguais em ambas. (Na realidade, o compilador verifica somente se um nome em duas estruturas diferentes tem o mesmo tipo e deslocamento em ambas, mas se os membros precedentes diferem, a construção não é transportável.)

Um exemplo simples de declaração de estrutura é

```
struct nodo {
    char palavra [20];
    int contador;
    struct nodo *esquerda;
    struct nodo *direita;
};
```

que contém um arranjo de 20 caracteres, um inteiro, e dois apontadores para estruturas similares. Uma vez dada essa declaração, a declaração

```
struct nodo s, *sp;
```

declara *s* como sendo uma estrutura do tipo dado e *sp* como sendo um apontador para uma estrutura do tipo dado. Com essas declarações, a expressão

```
sp -> contador
```

refere-se ao campo contador da estrutura apontada por *sp*;

```
s.esquerda
```

refere-se ao apontador da subárvore esquerda da estrutura *s*, e

```
s.direita -> palavra [0]
```

refere-se ao primeiro caractere do membro palavra da subárvore direita de *s*.

8.6 Inicialização

Um declarador pode especificar um valor inicial para o identificador sendo declarado. O inicializador é precedido por `=`, e consiste de uma expressão ou uma lista de valores entre chaves.

inicializador:

= expressão

= { lista-de-inicializadores }

= { lista-de-inicializadores , }

lista-de-inicializadores:

expressão

lista-de-inicializadores, lista-de-inicializadores

{ lista-de-inicializadores }

Todas as expressões num inicializador para uma variável estática ou externa devem ser expressões constantes, descritas na seção 15, ou expressões que se reduzem ao endereço de uma variável previamente declarada, possivelmente deslocada por uma expressão constante. Variáveis automáticas ou do tipo registrador podem ser inicializadas por expressões arbitrárias envolvendo constantes, e variáveis ou funções previamente declaradas.

Variáveis estáticas e externas que não são inicializadas contêm inicialmente o valor 0; variáveis automáticas e do tipo registrador não inicializadas contêm lixo.

Quando um inicializador se aplica a um *escalar* (um apontador ou um objeto de tipo aritmético), ele consiste de uma expressão única, talvez entre chaves. O valor inicial do objeto é tomado da expressão; as mesmas conversões aplicáveis à atribuição são executadas.

Quando a variável declarada é um *agregado* (estrutura ou arranjo), o inicializador consiste de uma lista de inicializadores, separados por vírgula e entre chaves, para os membros do agregado, escritos na ordem dos membros ou ordem crescente de índice. Se o agregado contém subagregados, essa regra se aplica recursivamente aos membros do subagregado. Se há menos inicializadores na lista que membros no agregado, o resto é preenchido com zeros. Não é permitido inicializar uniões ou agregados automáticos.

Chaves podem ser eliminadas como se segue. Se os inicializadores começam com uma chave esquerda, então a lista de inicializadores separados por vírgula que se segue inicializa os membros do agregado; é errado haver mais inicializadores do que membros. Se, entretanto, os inicializadores não começam com uma chave esquerda, então somente um número suficiente de elementos da lista são usados para inicializar os membros do agregado; quaisquer inicializadores restantes são deixados para inicializar o próximo membro do agregado corrente do qual faz parte.

Uma abreviação final permite que um arranjo de char seja inicializado por uma cadeia. Neste caso, caracteres sucessivos da cadeia inicializam os membros do arranjo.

Por exemplo,

```
int x [ ] = { 1, 3, 5 };
```

declara e inicializa x como um arranjo unidimensional que tem três membros, já que o tamanho não foi especificado e que há três inicializadores.

```
float y [4][3] = {  
    { 1, 3, 5 },  
    { 2, 4, 6 },  
    { 3, 5, 7 },  
};
```

é uma inicialização com chaves completas: 1, 3 e 5 inicializam a primeira linha do arranjo y [0], isto é, y [0] [0], y [0] [1] e y [0] [2]. Da mesma forma as próximas duas linhas inicializam y [1] e y [2]. Os inicializadores terminam antes do fim do arranjo e y [3] é inicializado com 0. O mesmo efeito poderia ser obtido com

```
float y [4] [3] = {  
    1, 3, 5, 2, 4, 6, 3, 5, 7  
};
```

O inicializador para y começa com uma chave esquerda mas isto não é o caso para y [0]; assim três elementos da lista são usados. Os próximos três elementos são usados para y [1] e mais três para y [2]. Também,

```
float y [4] [3] = {  
    { 1 }, { 2 }, { 3 }, { 4 }  
};
```

inicializa a primeira coluna de y (considerado como sendo um arranjo bidimensional) e deixa o resto com 0. Finalmente,

```
char msg [ ] = "Erro de sintaxe na linha %s\n";
```

mostra um arranjo de caracteres cujos membros são inicializados com uma cadeia.

8.7 Nomes de Tipos

Em dois contextos (para especificar conversões de tipos explicitamente por meio de um molde, e como argumento de sizeof), é desejável fornecer o nome de um tipo de dado. Isso é feito usando-se um “nome de tipo”, que, em essência, é uma declaração para um objeto desse tipo que omite o nome do objeto.

nome-de-tipo:

espec-tipo declarador-abstrato

declarador-abstrato:

vazio

(declarador-abstrato)

**declarador-abstrato*

declarador-abstrato ()

declarador-abstrato [expressao-constante @]

Para evitar ambiguidade, na construção

(declarador-abstrato)

o declarador abstrato não pode ser vazio. Sob essa restrição, é possível identificar univocamente a localização no declarador-abstrato onde o identificador apareceria se a construção fosse um declarador em uma declaração. O tipo referenciado é então o do identificador hipotético. Por exemplo,

```
int
int *
int * [3]
int (*) [3]
int * ( )
int (*) ( )
```

identificam respectivamente os tipos “inteiro”, “apontador para inteiro”, “arranjo de 3 apontadores para inteiro”, “apontador para um arranjo de 3 inteiros”, “função retornando apontador para inteiro”, e “apontador para função retornando inteiro”.

8.8 Typedef

Declarações cujas “classes de armazenamento” são typedef não definem armazenamento, mas definem identificadores que podem ser usados posteriormente como se fossem palavras-chaves de tipo dando um nome a tipos fundamentais ou derivados.

nome-typedef:

identificador

No escopo de uma declaração envolvendo typedef, cada identificador aparecendo como parte de qualquer declarador da declaração torna-se sintaticamente equivalente à palavra-chave associada ao identificador como descrito na subseção 8.4. Por exemplo, após

```
typedef int MILHAS, *AMEDIDA;
typedef struct {double real, imag;} complexo;
```

as construções

```
MILHAS distancia;
```

```
extern AMEDIDA ap_metro;
complexo z, *az;
```

são declarações legais; o tipo de distancia é int, o de ap_metro é "apontador para int", e de z é a estrutura especificada. az é um apontador para tal estrutura.

typedef não introduz novos tipos, mas somente sinônimos para tipos que poderiam ser especificados de outra forma. Assim, no exemplo acima, distancia é considerada como tendo o mesmo tipo de qualquer outro objeto de tipo int.

9. Comandos

Exceto quando indicado, comandos são executados em sequência.

9.1 Comando de Expressão

Muitos comandos são comandos de expressão, que têm a forma

expressao;

Comumente, comandos de expressão são atribuições ou chamadas de função.

9.2 Comando Composto ou Bloco

Para que vários comandos possam ser usados onde somente um é esperado, o comando composto (também chamado de "bloco") é fornecido:

comando-composto:

{ lista-de-declaracao @ lista-de-comando }

lista-de-declaracao:

declaracao

declaracao lista-de-declaracao

lista-de-comando:

comando

comando lista-de-comando

Se qualquer um dos identificadores na lista de declaração foi declarado previamente, a declaração anterior é empilhada durante a duração do bloco, após o qual ela reaparece.

Quaisquer inicializações de variáveis do tipo auto ou register são feitas a cada vez que se entra no início do bloco. É possível, atualmente (mas não é boa prática), transferir o fluxo de controle para dentro de um bloco; neste caso, as inicializações não são feitas. Inicializações de variáveis do tipo static são feitas apenas uma vez quando o programa começa. Dentro de um bloco, declarações do tipo extern não reservam armazenamento, de forma que não é possível inicializar tais variáveis.

9.3 Comando Condicional

As duas formas do comando condicional são

if (expressao) comando

if (expressao) comando else comando

Em ambos os casos a expressão é avaliada e, se é diferente de zero, o primeiro subcomando é executado. No segundo caso o segundo subcomando é executado se a expressão é 0; a ambiguidade do else é resolvida pela conexão de um else com o último if sem else.

9.4 Comando While

O comando while tem a forma

while (expressao) comando

o subcomando é executado repetidamente enquanto o valor da expressão não for zero. O teste é feito antes de cada execução do comando.

9.5 Comando Do

O comando do tem a forma

do comando while (expressao);

O subcomando é executado repetidamente até que o valor da expressão se torne zero. O teste é feito após cada execução do comando.

9.6 Comando For

O comando for tem a forma

for (expressao-1 @; expressao-2@; expressao-3@) comando

O comando é equivalente a

```
expressao-1;  
while (expressao-2) {  
    comando  
    expressao-3;  
}
```

A primeira expressão especifica a inicialização do laço; a segunda especifica um teste feito antes de cada iteração, e o laço é encerrado quando a expressão torna-se zero; a terceira expressão especifica, frequentemente, um incremento que é feito após cada iteração.

Qualquer uma, ou todas as expressões podem ser omitidas. A ausência da *expressão-2* faz com que o laço while equivalente seja while (1); outras omissões são simplesmente descartadas da expansão acima.

9.7 Comando Switch

O comando switch provoca a transferência de controle para um dentre vários comandos, dependendo do valor de uma expressão. Tem a forma

switch (expressao) comando

A conversão aritmética normal é executada na expressão, mas o resultado deve ser do tipo int. O comando é tipicamente composto. Qualquer comando dentro de comando pode ser rotulado com um ou mais prefixos como segue:

case expressao-constante:

onde a expressão constante deve ser do tipo int. Duas constantes de case num mesmo switch não podem ter o mesmo valor. Expressões constantes são definidas com precisão na seção 15.

Pode haver, também, no máximo prefixo de comando da forma

default:

Quando o comando switch é executado, sua expressão é avaliada e comparada com cada

constante `case`. Se uma das constantes é igual ao valor da expressão, o controle é passado para o comando que segue o `case`. Se nenhuma constante `case` é igual ao valor da expressão, e se há um prefixo `default`, o controle é passado para o comando com esse prefixo. Se não há prefixo `default`, nenhum comando do `switch` é executado.

`case` e `default` não alteram por si só o fluxo de controle, o qual continua livremente, atravessando tais prefixos. Para sair de um `switch`, vide `break`, na subseção 9.8.

Normalmente, o comando que é o objeto de um `switch` é composto. Declarações podem aparecer no começo desse comando, mas a inicialização de variáveis automáticas ou do tipo registrador não tem efeito.

9.8 Comando Break

O comando

```
break;
```

provoca o término do comando `while`, `do`, `for`, ou `switch` mais interno; o controle passa para o comando seguinte ao comando terminado.

9.9 Comando Continue

O comando

```
continue;
```

provoca a passagem do controle para a parte de continuação do comando `while`, `do` ou `for` mais interno; isto é, para o fim do laço. Mais precisamente, em cada um dos comandos

<pre>while (...) { ... contin; ; }</pre>	<pre>do { ... contin; ; } while (...)</pre>	<pre>for (...) { ... contin; ; }</pre>
--	---	--

um comando `continue` é equivalente a um `goto contin`. (O ; após `contin` é um comando nulo, vide a subseção 9.13.)

9.10 Comando Return

Uma função retorna ao seu chamador com o comando `return`, que tem uma das duas formas

```
return;  
return expressao;
```

No primeiro caso, o valor retornado é indefinido. No segundo caso, o valor da expressão é retornado ao chamador da função. Se necessário, a expressão é convertida, como em uma atribuição, para o tipo da função. Alcançar o final de uma função é equivalente a um `return` sem valor.

9.11 Comando Goto

O controle pode ser passado incondicionalmente por meio do comando

```
goto identificador;
```

O `identificador` deve ser um rótulo (veja subseção 9.12) na função corrente.

9.12 Comando Rotulado

Qualquer comando pode ser precedido por um rótulo da forma
identificador:

que serve para declarar o identificador como um rótulo. O único uso desse rótulo é servir como um alvo para um goto. O escopo de um rótulo é a função corrente, excluindo qualquer sub-bloco em que o mesmo identificador foi declarado. Veja a seção 11.

9.13 Comando Nulo

O comando nulo tem a forma

Um comando nulo é útil para permitir um rótulo antes de um fechamento de um comando composto, ou para fornecer um corpo nulo para um laço tal como um while.

10. Definições Externas

Um programa C consiste de uma sequência de definições externas. Uma definição externa declara um identificador tendo classe de armazenamento extern (por default) ou talvez static, e especifica um tipo. O especificador de tipo (subseção 8.2) pode ser vazio, e, neste caso, o tipo assumido é int. O escopo de definições externas persiste até o fim do arquivo onde elas são declaradas, da mesma forma que declarações persistem até o fim de um bloco. A sintaxe das definições externas é a mesma de todas as outras declarações, exceto que o código para funções só pode ser dado neste nível.

10.1 Definições de Funções Externas

Definições de funções têm a forma

definicao-de-funcao:

espec-de-decl @declarador-de-funcao corpo-de-funcao

Os únicos *espec-de-decl* permitidos entre os *espec-de-decl* são *extern* ou *static*; vide subseção 11.2 para distingui-los. Um *declarador de função* é similar a um *declarador de uma "função retornando ..."*, exceto que ele lista os parâmetros formais da função sendo definida.

declarador-de-funcao:

declarador (lista-de-parâmetros@)

lista-de-parametros:

identificador

identificador, lista-de-parametros

O corpo da função tem a forma:

corpo-de-funcao:

lista-de-declaracao comando-composto

Os identificadores na lista de parâmetros, e somente eles, podem ser declarados na lista de declaração. O tipo não especificado de quaisquer identificadores é assumido ser *int*. A única classe de armazenamento que pode ser especificada é *register*; se for especificada, o parâmetro atual correspondente será copiado, se for possível, num registrador no início da função.

Um exemplo de uma definição completa de uma função segue

```
int max (a, b, c)
int a, b, c;
{
    int m;

    m = (a > b) ? a : b;
    return ((m > c) ? m : c);
}
```

Aqui int é o espec-tipo;

max (a, b, c)

é o declarador-de-função;

int a, b, c;

é a lista-de-declaração para os parâmetros formais; { . . . } é o bloco fornecendo o código para o comando.

C converte todo parâmetro atual do tipo float para o tipo double, de modo que parâmetros formais declarados com float têm suas declarações ajustadas para double. Também, já que uma referência para um arranjo em qualquer contexto (em particular como um parâmetro atual) é considerada como sendo um apontador para o primeiro elemento do arranjo, declarações de parâmetros formais declarados como sendo "arranjo de . . ." são ajustadas para "apontador para . . .". Finalmente, já que as estruturas, uniões e funções não podem ser passadas para uma função, é inútil declarar um parâmetro formal como estrutura, união ou função (apontadores para tais objetos são permitidos, evidentemente).

10.2 Definições de Dados Externos

Uma definição de dado externo tem a forma:

definicao-de-dado:
declaracao

A classe de armazenamento de tal dado deve ser extern (que é o default) ou static, mas não auto ou register.

11. Regras de Escopo

Um programa C não precisa ser todo compilado ao mesmo tempo: o texto fonte do programa pode estar em vários arquivos, e rotinas pré-compiladas podem ser carregadas a partir de bibliotecas. A comunicação entre as funções de um programa pode ser feita por chamadas explícitas de funções e pela manipulação de dados externos.

Assim, há dois tipos de escopo a considerar: primeiro, o que pode ser chamado de *escopo léxico* de um identificador, que é essencialmente a região do programa na qual ele pode ser usado sem produzir diagnósticos de "identificador indefinido"; e o segundo, o escopo associado com identificadores externos, que é caracterizado pela regra que diz que referências ao mesmo identificador externo são referências ao mesmo objeto.

11.1 Escopo Léxico

O escopo léxico de identificadores declarados em definições externas vai da definição até o fim do arquivo-fonte em que elas aparecem. O escopo léxico de identificadores que são parâmetros formais é a função à qual estão associados. O escopo léxico de identificadores declarados no início de um bloco vai até o fim do bloco. O escopo léxico de rótulos é toda a função onde aparecem.

Desde que todas as referências ao mesmo identificador externo referem-se ao mesmo objeto (vide subseção 11.2), o compilador verifica se todas as declarações do mesmo identificador externo estão compatíveis; isto é, seu corpo é, na realidade, o arquivo inteiro onde aparecem.

Em todos os casos, entretanto, se um identificador é declarado explicitamente no início de um bloco, incluindo o bloco que constitui uma função, qualquer declaração desse identificador fora do bloco é suspensa até o fim do bloco.

Lembre-se, também, (subseção 8.5) que, por um lado, identificadores associados com variáveis ordinárias e por outro, aqueles associados com membros de estruturas e uniões e etiquetas, formam duas classes disjuntas que não conflitam. Membros e etiquetas seguem as mesmas regras de escopo dos outros identificadores. Nomes `typedef` estão na mesma classe dos identificadores ordinários. Eles podem ser redeclarados em blocos internos, mas um tipo explícito deve ser dado na declaração interna:

```
typedef float distancia;  
...  
{  
    auto int distancia;  
    ...  
}
```

O `int` deve estar presente na segunda declaração, ou ela seria considerada como sendo uma declaração sem declaradores com tipo `distancia`.*

11.2 Escopo de Objetos Externos

Se uma função refere-se a um identificador declarado como sendo `extern`, então em algum lugar nos arquivos ou bibliotecas que constituem o programa completo, deve haver uma definição externa para o mesmo. Todas as funções em um dado programa que se referem ao mesmo identificador externo, referem-se ao mesmo objeto, de modo que deve ser tomado cuidado para que o tipo e o tamanho especificados na definição sejam compatíveis com os especificados em cada função que referencia o dado.

A ocorrência da palavra `extern` numa definição externa indica que o armazenamento para os identificadores sendo declarados será alocado em outro arquivo. Assim, num programa em múltiplos arquivos, uma definição de cada objeto externo sem o especificador `extern` deve aparecer exatamente em um arquivo. Quaisquer outros arquivos que desejem ter uma definição externa do identificador devem incluir `extern` na definição. O identificador pode ser inicializado somente na declaração onde o armazenamento é alocado.

Identificadores declarados como sendo `static` em definições externas não são visíveis em outros arquivos. Funções podem ser declaradas `static`.

* Concordamos que a existência dessa regra é discutível.

12. Linhas de Controle do Compilador

O compilador C contém um pré-processador capaz de fazer a substituição de macro, compilação condicional, e a inclusão de arquivos. Linhas começando com # são destinadas ao pré-processador. Essas linhas têm sintaxe independente do resto da linguagem; elas podem aparecer em qualquer lugar e têm efeito (independente de escopo) até o fim do arquivo do programa fonte.

12.1 Reposição de Símbolos

Uma linha de controle do compilador da forma

```
#define identificador cadeia-de-símbolos
```

(note: não há ponto-e-vírgula no fim) faz o pré-processador mudar toda ocorrência subsequente do identificador pela cadeia de símbolos dada. Uma linha da forma

```
#define identificador (identificador, . . . , identificador) cadeia-de-símbolos
```

onde não há espaço entre o primeiro identificador e o abre-parênteses, é uma definição de macro com argumentos. Ocorrências subsequentes do primeiro identificador seguido por (, uma sequência de símbolos separados por vírgula, e), são repostas pela cadeia de símbolos da definição. Cada ocorrência de um identificador mencionado na lista de parâmetros formais da definição é repostada pela cadeia de símbolos correspondente da chamada. Os argumentos atuais na chamada são cadeias de símbolos separados por vírgula; entretanto, vírgulas em cadeias entre aspas ou entre parênteses não separam argumentos. O número de parâmetros formais e atuais deve ser o mesmo. O texto dentro de uma cadeia ou uma constante do tipo caractere não está sujeito a reposição.

Em ambas as formas, a cadeia de reposição é reexaminada procurando-se mais identificadores definidos. Em ambas as formas, uma definição longa pode ser continuada em outra linha escrevendo-se \no final da linha a ser continuada.

Essa facilidade é muito valiosa para a definição de “constantes simbólicas”, como em:

```
#define TAMTAB 100  
int tabela [TAMTAB];
```

Uma linha de controle da forma

```
#undef identificador
```

faz o pré-processador se esquecer da definição do identificador.

12.2 Inclusão de Arquivo

Uma linha de controle do compilador da forma

```
#include “nome-de-arquivo”
```

faz com que a linha seja repostada pelo conteúdo do arquivo dado. O arquivo é pesquisado inicialmente no diretório do arquivo-fonte, e depois em lugares padrão. Uma linha de controle da forma

```
#include <nome-de-arquivo>
```

pesquisa somente os locais padrão, e não o diretório do arquivo-fonte.

```
#include's podem ser não aninhados.
```


12.3 Compilação Condicional

Uma linha de controle do compilador da forma

if expressão-constante

verifica se a expressão constante (vide seção 15) é diferente de zero. Uma linha de controle da forma

ifdef identificador

verifica se o identificador está definido para o pré-processador; isto é, verifica se ele já foi objeto de uma linha de controle *# define*. Uma linha de controle da forma

ifndef identificador

verifica se o identificador não está definido para o pré-processador.

Todas as três formas são seguidas por um número arbitrário de linhas, possivelmente contendo uma linha de controle

else

e então por uma linha de controle

endif

Se a condição verificada é verdadeira, as linhas entre *# else* e *# endif* são ignoradas. Se a condição verificada é falsa as linhas entre o teste e o *# else* ou, na falta dele, o *# endif*, são ignoradas.

Essas construções podem estar aninhadas.

12.4 Controle de Linha

Para benefício de outros pré-processadores que geram programas C, uma linha da forma

line constante identificador

faz o compilador acreditar, para diagnósticos de erro, que o número da linha da próxima linha fonte é dado pela constante e o arquivo de entrada corrente é o indicado pelo identificador. Se o identificador é omitido, o nome do arquivo corrente não é alterado.

13 Declarações Implícitas

Não é sempre necessário especificar a classe de armazenamento e o tipo de identificador em declarações. A classe de armazenamento é fornecida pelo contexto em definições externas e em declarações de parâmetros formais e de membros de estruturas. Numa declaração dentro de uma função, se a classe de armazenamento é dada, mas não o tipo, o identificador é assumido ser do tipo *int*; se um tipo é dado, mas não a classe de armazenamento, o identificador é assumido ser da classe *auto*. Uma exceção à última regra aplica-se a funções, desde que funções *auto* não têm sentido (C seria incapaz de compilar código em uma pilha); se o tipo de um identificador é “função retornando . . .”, ele é implicitamente declarado como sendo *extern*.

Numa expressão, um identificador seguido por (e ainda não declarado é contextualmente declarado como “função retornando *int*”.

14. Tipos Reconsiderados

Esta seção resume as operações que podem ser feitas em objetos de certos tipos.

14.1 Estruturas e Uniões

Há somente duas coisas que podem ser feitas com uma estrutura ou união: referenciar um de seus membros (por meio do operador `.`); ou obter seu endereço (pelo operador unário `&`). Outras operações, tais como atribuição ou passagem como parâmetro, geram uma mensagem de erro. No futuro, espera-se que tais operações, mas não necessariamente outras, sejam permitidas.

Na subseção 7.1, diz-se que, numa referência direta ou indireta a uma estrutura (com `.` ou `->`), o nome à direita deve ser um membro da estrutura referenciada ou apontada pela expressão à esquerda. Para permitir fugir às regras de tipos, essa restrição não é rigidamente observada pelo compilador. De fato, qualquer lvalue é aceito antes do `.`, e é assumido como tendo a forma da estrutura de quem o nome à direita é membro. Também a expressão antes de `->` pode ser somente um apontador ou um inteiro. Se for um apontador, presume-se que ela aponta para uma estrutura de quem o nome à direita é membro. Se for um inteiro, é considerada como o endereço absoluto, em unidades de armazenamento da máquina, da estrutura apropriada.

Tais construções não são transportáveis.

14.2 Funções

Há somente duas coisas que podem ser feitas com uma função: chamá-la, ou obter seu endereço. Se o nome de uma função aparece numa expressão fora da posição do nome da função numa chamada, é gerado um apontador para a função. Assim, para passar uma função para outra, poder-se-ia dizer

```
int f ( );  
...  
g (f);
```

A definição de `g` poderia ser então

```
g (ap_func)  
int (*ap_func) ( );  
{  
    ...  
    (*ap_func) ( );  
    ...  
}
```

Observe que `f` deve ser declarada explicitamente na rotina chamadora visto que sua ocorrência em `g (f)` não foi seguida por `(`.

14.3 Arranjos, Apontadores e Indexação

Cada vez que um identificador do tipo arranjo aparece numa expressão, ele é convertido num apontador para o primeiro elemento do arranjo. Devido a essa conversão, arranjos não são lvalues. Por definição, o operador de indexação `[ ]` é interpretado de tal forma que `E1 [E2]` é idêntico a `*((E1) + (E2))`. Devido às regras de conversão aplicáveis ao operador `+`, se `E1` é um arranjo e `E2` um inteiro, então `E1 [E2]` refere-se ao `E2`-ésimo elemento de `E1`. Dessa forma, apesar de sua aparência assimétrica, a indexação é uma operação comutativa.

Uma regra consistente é seguida no caso de arranjos multidimensionais. Se E é um arranjo n -dimensional de dimensões $i_1 \times i_2 \times \dots \times i_n$, então E , aparecendo numa expressão, é convertido para um apontador para um arranjo $(n-1)$ -dimensional com dimensões $i_1 \times \dots \times i_{n-1}$. Se o operador $*$, explícita ou implicitamente como resultado de indexação, é aplicado a esse apontador, o resultado é o arranjo $(n-1)$ -dimensional sendo apontado, o qual é imediatamente convertido em um apontador.

Por exemplo, considere

```
int x [3] [5];
```

Aqui x é um arranjo 3×5 de inteiros. Quando x aparece numa expressão, ele é convertido num apontador para o primeiro dos três arranjos de cinco inteiros. Na expressão $x[i]$, o qual é equivalente a $*(x + i)$, x primeiro é convertido num apontador; depois i é convertido para o tipo de x , o que implica na multiplicação de i pelo tamanho do objeto para o qual o apontador aponta, isto é, referenciando 5 objetos inteiros. Os resultados são somados e uma indireção é aplicada para produzir um arranjo (de 5 inteiros) que é então convertido para um apontador para o primeiro dos inteiros. Se há outro índice, o mesmo argumento se aplica novamente; desta vez o resultado é um inteiro.

Conclui-se dessa discussão que arranjos em C são armazenados por linha (o último índice varia mais rapidamente) e que o primeiro índice na declaração ajuda a determinar a quantidade de armazenamento consumida pelo arranjo, mas não toma parte em qualquer outro cálculo de índices.

14.4 Conversões Explícitas de Apontadores

Certas conversões envolvendo apontadores são permitidas mas têm aspectos dependentes de implementação. Todas elas são especificadas por meio de um operador explícito de conversão de tipo. Vide as subseções 7.2 e 8.7.

Um apontador pode ser convertido para qualquer tipo integral com espaço suficiente para contê-lo. Se um `int` ou `long` é necessário depende da máquina. A função de mapeamento é também dependente de máquina, mas não deve surpreender aqueles que conhecem a estrutura de endereçamento da máquina. Detalhes de algumas máquinas particulares são dados abaixo.

Um objeto de tipo integral pode ser convertido explicitamente num apontador. O mapeamento sempre transforma um inteiro convertido a partir de um apontador de volta ao mesmo apontador, mas, além dessa regra, o mapeamento é dependente de máquina.

Um apontador para um tipo pode ser convertido para um apontador de outro tipo. O apontador resultante pode causar exceções de endereçamento se o objeto apontado não se refere a um objeto apropriadamente alinhado. É garantido que um apontador para um objeto de um dado tamanho pode ser convertido para um apontador de um objeto de menor tamanho e ser convertido de volta sem mudanças.

Por exemplo, uma rotina de alocação de armazenamento poderia aceitar um tamanho (em bytes) de um objeto a alocar, e retornar um apontador para char; ela poderia ser usada da seguinte forma.

```
extern char *aloca ( );
double *ad;
ad = (double *) aloca (sizeof (double));
*ad = 22.0/7.0;
```

aloca deve garantir (de forma dependente de máquina) que retornará um valor apropriado para conversão para um apontador para double; então o uso de função é transportável.

A representação de apontadores no PDP-11 corresponde a um inteiro de 16 bits e é medido em bytes. char's não têm restrições de alinhamento; o resto deve ter um endereço par.

No Honeywell 6000, um apontador corresponde a um inteiro de 36 bits; a parte da palavra está nos 18 bits à esquerda, e os 2 bits que selecionam o caractere na palavra estão imediatamente à sua direita. Então apontadores para char são medidos em unidades de 2 elevado a 16 bytes; o resto é medido em unidades de 2 elevado a 18 palavras de máquina. Quantidades double e agregados que os contêm devem se alinhar num endereço de palavra par (0 mod (2 elevado a 19)).

O IBM 370 e o Interdata 8/32 são similares. Em ambos, endereços são medidos em bytes; objetos elementares devem ser alinhados em fronteiras iguais às dos seus tamanhos, de forma que, apontadores para short devem ser 0 mod 2, para int e float 0 mod 4, e para double 0 mod 8. Agregados são alinhados no limite mais restritivo necessário por qualquer um dos seus constituintes.

15. Expressões Constantes

Em vários lugares C requer expressões constantes: após um case, como limites de arranjo, e em inicializações. Nos primeiros dois casos, as expressões podem envolver apenas constantes inteiras, constantes do tipo caractere, e expressões sizeof, possivelmente conectados pelos operadores binários

+ - * / % & | ^ << >> == != < > <= >=

ou pelos operadores unários

- ~

ou pelo operador ternário ? :

Parênteses podem ser usados para agrupar, mas não para chamadas de funções.

Mais liberdade é permitida para inicializadores; além de expressões constantes como discutidas acima, pode-se aplicar o operador unário & a objetos externos ou estáticos, e a arranjos externos ou estáticos indexados com uma expressão constante. O operador unário & também pode ser aplicado implicitamente pela ocorrência de arranjos sem indexação e funções. A regra básica é que inicializadores devem ser tanto uma constante ou o endereço de um objeto previamente declarado externo ou estático mais ou menos uma constante.

16. Considerações de Transportabilidade

Certas partes de C são inerentemente dependentes de máquina. A lista seguinte de problemas potenciais não é completa, mas indica os principais.

Características de hardware tais como o tamanho da palavra, propriedades da aritmética de ponto flutuante, divisão inteira, etc., têm provado na prática não ser um grande problema. Outras facetas do hardware são refletidas em diferentes implementações. Algumas dessas, particularmente a extensão de sinal (converter um caractere negativo num inteiro negativo) e a ordem em que os bytes são colocados numa palavra, são proble-

mas que devem ser cuidadosamente estudados. A maioria das outras diferenças de hardware representa problemas menores.

O número de variáveis da classe register que podem ser colocadas em registradores varia de máquina para máquina, assim como o conjunto de tipos válidos. No entanto, os compiladores fazem tudo de modo apropriado para suas máquinas; declarações register em excesso ou incorretas são ignoradas.

Algumas dificuldades aparecem somente quando práticas dúbias de codificação são usadas. É excessivamente imprevidente escrever programas que dependem de tais características.

A ordem de avaliação dos argumentos de funções não é especificada pela linguagem. É da direita para a esquerda no PDP-11, da esquerda para a direita nos outros. A ordem em que efeitos colaterais são executados também não é especificada.

Já que constantes do tipo caractere são realmente objetos do tipo int, constantes do tipo caractere envolvendo mais de um caractere são permitidas. A implementação específica é muito dependente de máquina, entretanto, porque a ordem em que os caracteres são atribuídos à palavra varia de máquina para máquina.

Campos são atribuídos a palavras, e caracteres a inteiros da direita para a esquerda no PDP-11 e da esquerda para a direita em outras máquinas. Essas diferenças são invisíveis para programas isolados que não convertem tipos explicitamente (por exemplo, pela conversão de um apontador de int para apontador de char e o acesso à memória apontada), mas devem ser consideradas no tratamento de formatos de memória impostos externamente.

A linguagem aceita pelos vários compiladores difere em pequenos detalhes. Mais notadamente, o compilador atual no PDP-11 não inicializa estruturas contendo campos de bits, e não aceita alguns poucos operadores de atribuição em certos contextos onde o valor da atribuição é usado.

17. Anacronismos

Como C é uma linguagem em evolução, certas construções obsoletas podem ser encontradas em programas antigos. Embora a maioria das versões do compilador suportem tais anacronismos, ultimamente eles têm desaparecido, deixando somente um problema de compatibilidade.

Versões anteriores de C usavam a forma `=op` ao invés de `op=` para operadores de atribuição. Isso levava à ambigüidade, como:

```
x=-1
```

que atualmente decremente `x` já que `=` e `-` são adjacentes, onde a intenção poderia facilmente ser a de atribuir `-1` a `x`.

A sintaxe de inicializadores foi mudada: anteriormente, o sinal de igual que introduz um inicializador não existia, de forma que ao invés de

```
int x = 1;
```

usava-se

```
int x 1;
```

A mudança foi feita porque a inicialização `int f (1 + 2)` assemelha-se o suficiente a uma declaração de função para confundir os compiladores.

18. Sumário da Sintaxe

Esse sumário da sintaxe de C objetiva mais auxiliar a compreensão do que ser uma apresentação exata da linguagem.

18.1 Expressões

As expressões básicas são:

expressao:
 primario
 * *expressao*
 & *expressao*
 - *expressao*
 ! *expressao*
 ~ *expressao*
 ++ *lvalue*
 -- *lvalue*
 lvalue ++
 lvalue --
 sizeof *expressao*
 (*nome-de-tipo*) *expressao*
 expressao op-bin *expressao*
 expressao ? *expressao* : *expressao*
 lvalue op-atr *expressao*
 expressao , *expressao*

primario:
 identificador
 constante
 cadeia
 (*expressao*)
 primario (*lista-de-expressao*@)
 primario [*expressao*]
 lavlue . *identificador*
 primario -> *identificador*

lvalue:
 identificador
 primario [*expressao*]
 lavlue . *identificador*
 primario -> *identificador*
 * *expressao*
 (*lvalue*)

Os operadores de expressão primária

() [] . ->

têm a maior prioridade e são associativos da esquerda para a direita. Os operadores unários

* & - ! ~ ++ -- sizeof (*nome-de-tipo*)

têm prioridade mais baixa que os operadores primários, porém mais alta que qualquer operador binário, e são associativos da direita para a esquerda. Operadores binários e o operador condicional são associativos da esquerda para a direita, e têm prioridade decrescente como indicado:

op-bin:

```
*      /      %
+      -
>>    <<
<      >    <=    >=
==     !=
&
^
|
&&
||
?:
```

Os operadores de atribuição têm a mesma prioridade, e são associativos da direita para a esquerda.

op-atr:

```
=      +=      -=      *=      /=      %=      >>=      <<=
&=     ^=      |=
```

O operador vírgula tem a prioridade mais baixa e é associativo da esquerda para a direita.

18.2 Declarações

declaração:

espec-de-decl lista-de-inic-de-decl@;

espec-de-decl:

espec-tipo espec-de-decl@

espec-ca espec-de-decl@

espec-ca:

```
auto
static
extern
register
typedef
```

espec-tipo:

```
char
short
int
long
unsigned
float
double
```

```

    espec-de-estr-ou-uniao
    nome-typedef

lista-de-inic-de-decl:
    declarador-inic
    declarador-inic , lista-de-inic-de-decl

declarador-inic:
    declarador inicializador@

declarador:
    identificador
    (declarador)
    *declarador
    declarador ( )
    declarador [expressao-constante@]

espec-de-estr-ou-uniao:
    struct {lista-de-decl-de-estr}
    struct identificador {lista-de-decl-de-estr}
    struct identificador
    union {lista-de-decl-de-estr}
    union identificador {lista-de-decl-de-estr}
    union identificador

lista-de-decl-de-estr:
    decl-estr
    decl-estr lista-de-decl-de-estr

decl-estr:
    espec-tipo lista-declaradores-estr;

lista-declaradores-estr:
    declarador-estr
    declarador-estr , lista-declaradores-estr

declarador-estr:
    declarador
    declarador: expressao-constante
    : expressao-constante

inicializador:
    = expressao
    = { lista-de-inic }
    = { lista-de-inic, }

lista-de-inic:
    expressao

```


lista-de-inic , *lista-de-inic*
{ *lista-de-inic* }

nome-de-tipo:
espec-tipo *declarador-abstrato*

declarador-abstrato:
vazio
(*declarador-abstrato*)
* *declarador-abstrato*
declarador-abstrato ()
declarador-abstrato [*expressao-constante*@]

nome-typedef:
identificador

18.3 Comandos

comando-composto:
{ *lista-de-decl*@ *lista-de-comando*@ }

lista-de-decl:
declaracao
declaracao *lista-de-decl*

lista-de-comando:
comando
comando *lista-de-comando*

comando:
comando-composto
expressao;
if (*expressao*) *comando*
if (*expressao*) *comando* else *comando*
while (*expressao*) *comando*
do *comando* while (*expressao*);
for (*expressao-1*@ ; *expressao-2*@ ; *expressao-3*@) *comando*
switch (*expressao*) *comando*
case *expressao-constante*: *comando*
default: *comando*
break;
continue;
return;
return *expressao*;
goto *identificador*;
identificador: *comando*
;

18.4 Definições Externas

programa:
 definicao-externa
 definicao-externa programa

definicao-externa:
 definicao-de-funcao
 definicao-de-dados

definicao-de-funcao:
 espec-tipo@ declarador-de-funcao corpo-de-funcao

declarador-de-funcao:
 declarador (lista-de-parametros@)

lista-de-parametros:
 identificador
 identificador , lista-de-parametros

corpo-de-funcao:
 lista-de-decl-de-tipo comando-de-funcao

comando-de-funcao:
 { *lista-de-decl@ lista-de-comando* }

definicao-de-dado:
 extern@ espec-tipo@ lista-de-inic-de-decl@ ;
 static@ espec-tipo@ lista-de-inic-de-decl@ ;

18.5 Pré-processador

define identificador cadeia-de-simbolos
define identificador (identificador , . . . , identificador) cadeia-de-simbolos
undef identificador
include "nome-de-arquivo"
include < nome-de-arquivo >
if expressao-constante
ifdef identificador
ifndef identificador
else
endif
line constante identificador

ÍNDICE ANALÍTICO

- / caractere contrabarra 20-21, 43-44, 164
- \ caractere de escape 20-21, 43-44, 164
- \0 caractere nulo 38, 43-44
- ?: expressão condicional 54-55, 174-175
- + operador adicional 47-48, 171-172
- > operador apontador de estrutura 116, 168
- / operador binário 23, 45-46, 172
- & operador e bit-a-bit 52, 173-174
- ~ operador de complemento 53, 170-171
- operador de decremento 28, 50, 100, 170
- >> operador de deslocamento para direita 52-53, 172
- ^ operador ou-exclusivo bit-a-bit 52, 173-174
- = = operador igualdade 29, 46-47, 173-174
- | operador ou-inclusivo bit-a-bit 52, 173
- ++ operador de incremento 28, 39, 100, 170-171
- * operador indireção 89, 170-171
- && operador e lógico 31, 46, 52-53, 173
- || operador ou lógico 31, 46, 52-53, 174
- > operador "maior que" 46, 172
- > = operador "maior que ou igual" 46, 172
- % operador módulo 45, 171-172
- * operador de multiplicação 45, 171-172
- ! = operador "não igual a" 27, 46, 172
- ! operador de negação lógica 46, 170
- operador subtração 45, 171-172
- & operador unário 89, 170
- operador unário menos 45, 170
- agregado 181-182
- alinhamento de campo 130, 180
- alinhamento por *união* 160
- alocador de memória 95, 126, 129, 192-193
- alocador de memória *calloc* 145-146
- ambigüidade *IF-ELSE* 58-59, 184
- anacronismos 195
- ano bissexto, computação 45-46
- < *a.out* 20, 71-72
- apontador argumento 94-95
- apontador aritmético 89-90, 93-94, 110, 122-123, 172
- apontador aritmético ilegal 97, 98
- apontador aritmético, modificando em 93-94, 97
- apontador, arquivo 140-141, 152-153
- apontador, declaração de 89-90, 94-95, 177-178
- apontador para estrutura 121, 122
- apontador para função 110, 132-133, 183-184
- apontador, inicialização 96, 122-123
- apontador *NULL* 96, 175-176
- apontador com subtração 97
- apontadores 35-36
- apontadores e arranjos 92-93, 107
- apontadores, arranjo de 102-103
- apontadores e índices 92-93
- apontadores, operações permitidas 97-98
- argumento, apontador 94-95
- argumento contador *argc* 106-107
- argumento, número de variáveis do 74
- argumento de subarranjo 94-95

- argumentos # *define* 87-88
- argumentos de linha de comando 106-107
- argumentos de programa 106-107
- arquivo, acesso 140-141, 147
 - adição 141-142, 151-152
 - concatenação 140-141
 - criação 141-142, 147
 - modo de acesso 141-142, 149-150
 - programa de cópia 27-28, 148
- arquivo, *rewind* 152
- arquivos múltiplos, compilando 71-72
- arranjo de apontadores 102-103
- arranjo bidimensional 101
- arranjo bidimensional de Índices 102
- arranjo bidimensional, inicialização de 103
- arranjo de caractere 30, 36, 97-98
 - inicialização 85-86, 182-183
- arranjo explicação de Indexação 92-93, 193
- arranjo, inicialização 84-85, 106, 181-182
- arranjo multidimensional 101, 193
- arranjo, ordem de memória 102, 193
- arranjos e apontadores 92-93, 107
- arranjos de estruturas 117-118
 - inicialização de 119
- árvore binária 123
- asm*, palavra chave 164
- associatividade de operadores 56, 168
- atribuição, conversão por 49, 174-175
- atribuição, múltiplo 31
- automáticos, extensão dos 78-79
- automáticos, inicialização de 38-39, 46, 84, 181-182
- avaliação, ordem de 31, 46, 53, 57, 65, 78, 88, 90-91, 168, 195
-
- \ b, caractere retrocesso 20-21
- biblioteca padrão 92, 95, 121, 134, 147, 153
- bit, manipulação de idiomas 129-130
- bit-a-bit e operador & 51-52, 173-174
- bit-a-bit exclusivo ou operador, ^ 52, 174
- bit-a-bit inclusivo ou operador, | 52, 174
- bloco 58, 82-83, 184
- bloco, inicialização em 184
- byte 120, 171-172
-
- cachimbo 134-135, 148
-
- cadeia
 - abertura 140-141, 147
 - apontador 140-141, 153
 - inclusão 87, 190
 - modo de proteção 149-150
 - retrocesso 152
- cadeia de caracteres 20-21, 45, 81-82, 165
- cadeia, final de 38, 43-44
- cadeia, tamanho de 38, 45, 98
- cadeia, tipo de 120
- cadeia null 45
- campo de alinhamento 130, 180
- campos de bit 128-129, 179-180
- caractere aspa 21, 30-31, 44-45
- caractere assinalado 48, 165-166
- caractere constante octal 44
- caractere contrabarra 21, 44, 164
- caractere entrada\saída 26
- caractere, escape, \ 21, 44, 164
- caractere, funções de teste 145
- caractere negativo 166-167
- caractere nova-linha 21, 29-30
- caractere null 44-45
- caractere sublinha 42, 163
- caractere tabulação 21
- caractere único 29-30
- carga da biblioteca padrão 134-135
- carregando cadeias múltiplas 71-72
- cfree* 145-146
- chamada de função, semântica 164
- chamada de função, sintaxe 169-170
- chamada por referência 36, 74, 90-91, 169-170
-
- chamada de sistema *creat* 150
- chamada de sistema *lseek* 152
- chamada do sistema *open* 150
- chamada do sistema *sbrk* 160-161
- chamada do sistema *seek* 152
- chamada do sistema *stat* 156-157
- chamada do sistema *unlink* 151
- chamada do sistema *Write* 148
- chamada por valor 36, 74, 90-91, 169-170
- chaves 20, 22-23, 58
- chaves, posição das 24
- classe de memória *auto* 34-35, 39, 175-176
- classe de memória estática 39, 82, 175-176
- classe de memória, *externa* 175-176
- classe de memória, *register* 82, 175-176

- classe de memória, *static* 39, 82, 175-176
- comando de atribuição, aninhado 27-28, 31
- comando *break* 62, 66-67, 186
- comando *cc* 20, 71-72, 135
- comando composto 58, 83, 184
- comando *continue* 68, 186
- comando *do* 64-65, 185
- comando *# else* 31
- comando *for* 25, 29, 62, 185
- comando *goto* 67-68, 186-187
- comando *if* 30
- comando *if-else* 31, 58, 184
- comando *null* 28-29, 187
- comando rotulado 187
- comando */s* 156
- comando *switch* 33, 60, 76, 185
- comando vazio 29, 187
- comando *while* 22-23, 62, 185
- comandos 184
- comandos, sequência de 184
- comentário 21-23, 163
- comparação de apontadores 97, 173
- compilação condicional 191
- compilação, separado 69, 79, 188
- compilando arquivos múltiplos 71-72
- compilando um programa C 20, 34-35
- condição de limite 28-29, 30, 106
- conjunto de caracteres ASCII 30, 44, 48, 121
- conjunto de caracteres EBCDIC 44, 48
- constante cadeia 21, 31, 44, 98, 165
- constante caractere 30, 44, 164
- constante *double* 165
- constante flutuante 24, 165
- constante hexadecimal 44, 164
- constante integrante 24, 164
- constante *long* 44, 164
- constante octal 44, 164
- constante simbólica 25, 27, 31, 87, 121
- constante simbólica, tamanho das 42
- constante, tipo caractere 30, 44, 164
- constantes 164
- contenção 124
- convencões léxicas 163
- conversão apontador - integrante 100, 168, 172, 193
- conversão de apontadores 126, 172, 193
- conversão de argumento 50
- conversão por atribuição 49, 175
- conversão caractere - integrante 33, 48, 167
- conversão de caractere inteiro 33, 48, 167
- conversão de datas 101, 114, 116
- conversão *double float* 49, 73, 168
- conversão entrada 91-92
- conversão *float double* 49, 73, 168
- conversão flutuante - integrante 50, 168
- conversão formatada 24
- conversão de função 169
- conversão, função argumento 50-51, 164
- conversão integrante-apontador 171, 192-193
- conversão integrante-caractere 49, 167
- conversão, integrante-flutuante 23-24, 168
- conversão integrante-long 167
- conversão integrante-unsigned 168
- < conversão não-assinalado-inteiro 168
- conversão de nome de arranjo 93-94, 169
- conversão de tipo por *return* 74, 186
- conversão de tipo explícita 50, 126, 146
- conversão de valor real 49
- conversão unsigned-integrante 168
- conversão % 0 24-25
- conversão % l d 29
- conversão % s 24-25, 38
- conversão % d 11
- conversão % r 24-25
- conversões aritméticas 48, 168
- decisão múltipla 34, 60
- declaração de apontador 89-90, 94-95, 178
- declaração de arranjo 32, 179
- declaração, campo 130, 180
- declaração, classe de memória 176
- declaração de definição 78-79
- declaração estática 82
- declaração, estrutura 114, 180
- declaração extern 40-41
- declaração FILE 141
- declaração de função 178
- declaração de função, implícita 73, 170, 192
- declaração implícita 191
- declaração implícita de função 74
- declaração mandatória 21-23, 45, 78
- declaração register 83
- declaração, sintaxe de 45

- declaração *static* 82
- declaração tipo 178
- declaração tipo inconsistente 74
- declaração *typedef* 132, 176, 184
- declaração *union* 130, 180
- declaração de variável externa 39
- declarações 176
- declarador 177
- declarador abstrato 183
- default, inicialização 87, 106
- default, tamanho do arranjo 87, 106, 119, 179
- definição *extern* 39, 187
- definição, função 35, 71, 187
- definição de variável externa 41
- definições de dados 188
- definições *stat* 158
- dependência na máquina 47, 48, 53, 83, 97, 101, 123, 126, 130, 131, 133, 153, 159, 195
- descritores, arquivos 147
- deslocamento para esquerda, operador << 53, 173
- devolução de entrada 81
- diretório 156
- divisão, integrante 24, 46, 193

- efeitos colaterais 82, 88, 169
- eficiência 55, 69, 83, 87, 106, 121, 126, 134, 148, 162
- # *else* 191
- else-if* 33, 60
- endentação 24, 29, 59
- # *endif* 191
- endereço de variável 36, 89
- entrada com buffer 148
- entrada sem buffer 148
- entrada de caractere 26
- entrada, devolução de 81
- entrada formatada 25, 138
- entrada inode 156
- entrada liberada 91-92
- entrada padrão 135, 148
- entrada, redireção 135
- entrada e saída, terminal 26, 135, 148
- entry 164
- escopo de automáticos 79
- escopo léxico 188

- especificador de classe de memória 176, 177
- especificador de classe de memória omitida 177
- especificador de tipo 177
- especificador de tipo omitido 177
- estáticas, inicialização de 46, 84, 182
- estrutura, aninhada 115
- estrutura apontadora para 122
- estrutura auto-referencial 124, 181
- estrutura de bloco 58, 83, 184
- estrutura de diretório *dir. h* 156
- estrutura, etiqueta 114
- estrutura, operações permitidas em 192
- estrutura *stat* 157
- estruturas, arranjos de 118
- estruturas permitidas, operações 192
- estruturas, restrições em 116, 192
- etiqueta, estrutura 114, 181
- etiqueta, união 181
- exit-exit 144
- exponenciação 34
- expressão 169
- expressão, atribuição 28, 31, 55, 175
- expressão condicional, ?: 55, 175
- expressão, constante 44, 61, 194
- expressão, índice 32
- expressão, ordem de avaliação da 56, 169
- expressão com parênteses 170
- expressão primária 169
- expressão relacional, valor de 49
- expressão subscripta 32
- expressão, tipo de 25
- expressão unária 171
- extensão de cadeia 38, 45, 98
- extensão do campo 130
- externos, escopo de 79, 189
- externos, inicialização de 46, 75, 79, 84
- extensão de nomes 42, 163
- extensão do sinal 32, 49, 153, 195

- % *f* conversão 25, 29
- f close* 143
- f gets* 144
- fopen* 141
- f print f* 142
- f puts* 144
- f scanf* 142

final de arquivo 27, 135
 final de cadeia 38, 44
 forma permitida de inicializador 194
 fortran, palavra-chave 164
 função *ALOCA* 96, 161
 função, apontadores para 111, 133, 192
 função argumento 20, 35, 170
 função argumento, conversão 50, 170
 função *arvore* 125
 função *atof* 73
 função *atoi* 48, 64
 função de biblioteca 20, 21, 69, 74
 função *comprum* 113
 função *comprime* 51
 função *contabit* 55
 função, conversão de 169
 função *copy* 37
 função, declaração de 178
 função, declaração implícita de 72, 170, 192
 função, definição 35, 71, 187
 função *dia-do-ano* 102, 117
 função *fillbuf* 155
 função *fopen* 153-154
 função *getbits* 53
 função *getch* 81
 função *getint* 92-93
 função *getline* 37, 71
 função *gettop* 80
 função *getword* 121
 função *hash* 128
 função *impr-árvore* 126
 função *imprlinhas* 105, 106
 função *index* 71
 função *install* 129
 função *itoa* 66
 função *libera* 96, 162
 função *lookup* 128
 função *main* 20
 função *maismem* 162
 função *mês-dia* 102, 118
 função *mês-nome* 106
 função minúscula 48
 função *obtlinhas* 105
 funções, operações permitidas 192
 funções permitidas, operações 192
 função de pesquisa binária 60, 120, 123
 função *pop* 79

função *power* 35, 36
 função *printf* 87
 função *sort* 64, 106, 112
 função *stramp* 100
 função *strcpy* 98-99
 função *strcat* 52
 função *strlen* 45, 95, 97
 função *strsave* 101
 funções, teste de caractere 145
 função troca 91-92, 113
 função *type* 118
 função *ungetch* 81
 função vazia 71
 função, valor de retorno 35

getc 142
getc macro 153
getchar 26, 135
getchar, c/buffer 150
getchar, s/buffer 149
getchar macro 142
getchar, valor de 48
 agrupamento de comando 58

idioma de laço DO 63
#if 191
#ifdef 191
 if-else, ambigüidade 58, 184
#include 87, 134, 186
#ifndef 191
 index de arranjos bidimensionais 101
 informação, omissão 69, 70, 76, 78, 83
 inicialização 46, 182
 inicialização, apontador 96, 123
 inicialização, arranjo 85, 106, 182
 inicialização de arranjos bidimensionais 103
 inicialização de arranjos de estruturas 119
 inicialização de automáticos 39, 46, 84, 182
 inicialização em blocos 184
 inicialização de estáticos 46, 84, 182
 inicialização de estrutura 114, 116, 182
 inicialização de externos 46, 75, 79, 84
 inicialização de registros 84
 inicialização, forma permitida de 194
 índices e apontadores 93
 interdata UNIX 2

laço, *for* 25
 laço, infinito 63
 laço, teste na base 65
 laço, teste no topo 30, 65, 106
 laço, *while* 22-23
 legibilidade, programa 24, 28, 31, 42, 48,
 55, 59, 63, 66, 67, 85, 87, 133
 linguagem "sem tipo"
 linhas de texto, ordenação 103
 lista de argumento 20
 listagem de palavras-chave 164
 lógica E operador, && 31, 46, 53, 174
 lógica ou operador, || 31, 46, 53, 175
lvalue 183

macro

isalpha 121, 145
isdigit 121, 145
islower 145
isspace 145
isupper 135, 145

macro *toupper* 145

macros com argumentos 88

modificações em apontador aritmético 93,
 97

modo de proteção, arquivo 150

modularização 34, 36, 41, 75, 76, 104

nome de arranjo, argumentos 36, 74, 95,
 103, 188

nome de arranjo, conversão de 94, 169

nome do membro, estrutura 114, 181

nome de tipos 183

nomes externos, tamanho 42, 163

nomes de função, tamanho dos 42, 163

nomes internos, tamanho dos 42

nomes de variáveis, sintaxe 42

nomes de variáveis, tamanho 42

notação científica 43

notação E 43

notação infixada 75

notação Polonesa 75

notação polonesa reversa 75

notação sintática 166

número de argumentos, variável 74

número mágico 25

números, tamanho dos 23, 29, 43, 166

operações permitidas em apontadores 98

operações permitidas em estruturas 192

operações permitidas em funções 192

operações permitidas em uniões 192

operador ~ 53, 171

operador de adição + 46, 172

operador, apontador 117, 126, 169, 172,
 193

operador de atribuição 47, 54, 175

operador de conversão, explícito 171, 193

operador de conversão de tipo 50

operador de decremento, -- 29, 50, 100,
 171

operador de deslocamento para direita, >>
 53, 173

operador de endereço & 89, 171

operador de força 50

operador indireção, * 89, 171

operador de incremento, ++ 29, 50, 100,
 171

operador "maior que", > 47, 173

operador "maior que ou igual", >= 47,
 173

operador não-igualdade, != 27, 47, 174

operador de membro da estrutura 115, 169

operador "menor que ou igual", <= 47,
 173

operador de molde 50, 126, 146, 171, 183

operador de módulo, % 46, 172

operador de multiplicação, * 46, 172

operador de negação lógica, ! 47, 171

operador sizlof 120, 171

operador subtração, - 46, 172

operador unário menos, - 46, 171

operador vírgula 64, 176

operador aditivo 172

operador, aritmética 46, 172

operador, associatividade 56, 169

operador, atribuição 47, 54, 175

operador, bit-a-bit 52, 174

operador, comutativo 46, 57, 169

operador, deslocamento 52, 173

operador, igualdade 47, 174

operador lógico 47

operador multiplicador 169

operador, procedência dos 28, 56, 91, 116,
 117, 169, 197

operador, relacionais 27, 47, 173

ordem de arranjo de memória 104, 193
ordem de avaliação 31, 46, 53, 56, 57, 65,
78, 88, 91, 169, 195
ordem de avaliação de expressões 56, 169
ordenação lexicográfica 111
ordenação de linhas de texto 103
ordenação numérica 111
origem, índice 32

padrão de erro 148
palavras-chave, lista de 164
palavras reservadas 42, 167
parâmetro formal 35, 83, 95, 187
ponto-e-vírgula 23, 26, 29, 58, 60
portabilidade 44, 48, 53, 74, 101, 131, 133,
134, 136, 153, 159, 195
posição das chaves 24
prefixo *case* 61, 185
preprocessador C 87
preprocessador macro 87, 190
printf 20, 25, 53
procedência de operadores 28, 56, 91, 117,
118, 169, 197
programa calculador 74, 75, 78, 140
programa *cat* 141, 143, 144
programa, classificação 104, 112
programa de consulta à tabela 127
programa de contagem de caracteres 29
programa de contagem de espaços brancos
32, 62
programa de contagem de linha 30
programa de contagem de palavras 31, 123
programa de contagem de palavras-chave
120
programa conversão de temperatura 21-23
programa de cópia de arquivo 26, 148
programa *cp* 151
programa copy 27-28
programa *echo* 108
programa encontro de padrão 70, 109, 110
programa formatado 24, 29, 123, 163
programa *fsize* 158
programa *grep* 69, 109
programa legibilidade 31, 48, 63, 66, 67,
87
programa da letra minúscula 136
programa de listagem de diretório 156
programa maior linha 37

programa minúscula 136
programa de ordenação 104, 112
PDP-11 UNIX
programa, pesquisa em 127
programa remove brancos 67
putc 142
putc macro 153
putchar 26, 135
recursão 86, 124, 125, 170
redireção EIS 135, 142, 148
redireção de saída 135
referência estrutural, semântica 170
referência estrutural, sintática 170
registradores, inicialização de 84
regras de conversão 49
regras, conversão de tipo 49, 168
regras, escopo 78-79, 188
reposição de símbolos 190
restrição de alinhamento 123, 126, 132,
146, 159, 193
restrições em campos 130, 180-181
restrições em estruturas 116, 185
restrições em registros 195
restrições em tipos 179
restrições em uniões 132, 192
return, conversão por 74
return, conversão de tipo por 74, 186
rótulo 67-68, 187
saída formatada 23-24, 136
scanf 137-138
semântica, referência estrutural 170
seqüência de comandos 184
seqüência escape 20-21, 30, 45, 164
shell, algoritmo de ordenação 64, 106
sintaxe de declaração 45
sintaxe de nomes variáveis 42
sintaxe, referência de estrutura 169-170
sintaxe, resumo 196
sistema de arquivo UNIX
sistema *close*, chamada de 151
sistema *read*, chamada 148
sprintf 139-140
sscanf 139-140
stat.h 157
stderr 142, 144
stdin 142

stdio.h, arquivo padrão 134
stdio.h, conteúdo 148
stdout 142
 subtração, apontador 97
 supressão de atribuição, *scanf* 138
system 146

 tabela *hash* 127
 tamanho do arranjo, *default* 86, 106, 119, 179
 tamanho de números 23, 29, 42-43, 166
 terminal de comando 23, 58
 terminal entrada e saída 26, 134, 148
 terminação, programa 142, 144
 teste na base do laço 64-65
 teste no topo do laço 29-30, 64-65, 106
 tipo não-assinalado 42, 53, 167, 177
 tipo de cadeia 170
 tipo *char* 23, 42-43, 166, 177
 tipo *double* 23, 29, 42-43, 177
 tipo de expressão 25
 tipo *float* 23, 42-43, 177
 tipo inconsistente 73-74
 tipo *int* 23, 42-43, 177
 tipo *long* 23, 29, 42-43, 166, 177
 tipo *short* 23, 42-43, 166, 177
 tipo de uma variável 21-23
 tipos aritméticos 167
 tipos derivados 19, 22-23, 167
 tipos flutuantes 167
 tipos fundamentais 22-23, 166
 tipos integrais 167
 tipos, restrições em 178-179
 transbordo 64, 169
 truncamento 23-24, 46, 157

typedef, uso de 159
types.h 159

#undef 190
ungetc 145
 união, alinhamento por 160
 união, declaração 130, 180
 uniões permitidas, operações 192
 uniões, operações permitidas em 192
 uniões, restrições em 131, 192
 UNIX, PDP 11
 uso de *typedef* 159

valor de EOF 26-27, 48, 134-135
 valor de expressão lógica 49
 valor de expressão relacional 49
 valor de *getchar* 48
 variável, automática 34, 39, 75
 variável, endereço de 36, 89
 variável externa 39, 75
 variável externa, declaração de 39
 variável externa, definição de 40-41
 variável local 39
 variáveis estáticas, externas
 variáveis estáticas, internas 81-82
 variáveis externas *static* 81-82
 variáveis internas *static* 81-82
 verificação de tipo 125-126
 verificador de programa C (*lint*)
 verificador de programa *lint* 57, 71-72, 101, 133
 vetor argumento *argv* 106-107

 zero, teste omitido *c/* 59, 97-98, 99-100

Impressão e acabamento
(com filmes fornecidos):
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP



C

A LINGUAGEM DE PROGRAMAÇÃO

Este livro prepara o leitor para usar a linguagem C. Através de exemplos, são detalhadamente explicadas todas as características da linguagem. Os exemplos são programas completos, que não apenas ensinam a linguagem, mas também ilustram algoritmos, estruturas de dados e técnicas de programação importantes.

As principais características da linguagem C são a economia de expressão, a existência de controles de fluxo e estruturas de dados bastante modernos e um riquíssimo conjunto de operadores. Versátil, expressiva e eficiente, é a linguagem utilizada no sistema operacional UNIX, mas também pode ser utilizada em outros sistemas.

As principais rotinas de I/O são apresentadas, e o livro inclui ainda um útilíssimo MANUAL DE REFERÊNCIA C.

BRIAN W. KERNIGHAN

é PhD em Ciências da Computação pela Universidade de Princeton e membro do staff técnico do Computing Research Center dos laboratórios da Bell em Nova York.

DENNIS M. RITCHIE

é um dos responsáveis pelo desenvolvimento, na Bell, da linguagem C e do sistema operacional UNIX.

Conheça também da Editora Campus:

C: A Linguagem de Programação Padrão ANSI — Kernighan / Ritchie

ISBN 85-7001-410-4